Leistungsstark !

Flexibel !

Schnell !

Precision
Software Tools

Deutsche Ausgabe - GBXL

ACTION!-System, Run Time und Toolkit in einem Handbuch

Bearbeitungsstand: 20. Mai 2016

Deutsches Handbuch

zu

The Action! System

Die vollständige Programmierumgebung
für ATARI 8-Bit-Computer.

und

The Action!
Run Time Package

Die Compilierung zu einem unabhängig vom Modul lauffähigem Programm.

und

The Action! Toolkit

Das unverzichtbare Hilfspaket
für alle Action!-Programmierer.

Vorwort

Action! ist ungeschlagen die leistungsfähigste höhere Programmiersprache für ATARIs 8-Bit-Rechner und sowohl für Anfänger als auch für fortgeschrittene Programmierer ideal geeignet. Selbst größere Projekte lassen sich sehr gut in Action! realisieren[1].

Action! ist die schnellste Hochsprache für den A8 mit Geschwindigkeitsfaktor 1,5 in Relation zu Maschinensprache. Weder C noch Pascal oder Turbo BASIC XL erreichen annähernd derartige Werte in den Benchmarktests.

Dieses Handbuch richtet sich an deutschsprachige Anwender und soll ihnen einen leichteren Zugang zum Action!-Programmiersystem bieten.

Das ungebrochene Interesse an dieser Programmiersprache veranlasste mich zu einer neuen wesentlich erweiterten Auflage; hier also Action!, die Fünfte.

Das 'Action! Toolkit' und das 'Action! Run Time Package' wurden neu in dieses Handbuch eingearbeitet. Dabei flossen zusätzlich Hinweise aus der letzten bekannten Fehlerliste (3) mit ein, sofern sie die aktuellen Versionen von Action! (3.6), Action! Toolkit (3) und Action! Run Time Package (1.4) betreffen.

Mögen alle Nutzer dieses Handbuches das über Action! erfahren, was sie schon immer wissen wollten!

Eine Anmerkung zur rechtlichen Situation: Im Januar 2015 hat der Autor von Action!, Clinton W. Parker, den Quellcode und damit Action! an sich als Freeware freigegeben; nachzulesen hier:

http://atariage.com/forums/topic/217770-action-source-code/#entry3166744

Die Produkte von OSS und ICD wie Action!, BASIC XL, SpartaDOS X oder das MIO und viele mehr haben Maßstäbe gesetzt. Als Abandonware werden sie weiterhin engagiert genutzt, verbessert, erweitert und verfügbar gehalten durch die weltweite Interessengemeinschaft der ATARI-8-Bit-User.

Dieses Handbuch dient dem Erlernen von Action! sowie dem Erhalt dieser einzigartigen Programmiersprache.

Dank geht an Carsten, Jac und Erhard für Hinweise und Hilfen bei der Überarbeitung dieser Ausgabe sowie an Roland für die Schöpfung wesentlicher Quellen.

GoodByteXL im März 2015

---

1) Terminalprogramm in Mini Office II; SpartaDOS Sektor Editor 'DiskRx'.

Inhaltsverzeichnis

I. Einführung

1 Allgemeines

Action! ist ein komplettes Entwicklungssystem für A8-Software - ein System, das alle Programmierwünsche erfüllt. Wer bereits mit BASIC vertraut ist, wird feststellen, dass Action! viel schneller arbeitet und über einen besseren Editor verfügt, aber genauso leicht zu erlernen ist.

Assemblerfans wird es freuen, dass Action! in der Programmausführung fast so schnell wie Maschinensprache ist. Als Anwender merkt man alsbald, dass Programmieren in Action! aufgrund der Leistungsfähigkeit der Sprache, des Editors und der bereits vorhandenen Routinen zügig und leicht von der Hand geht. Dieses Buch ist ein Nachschlagewerk, in dem die Fähigkeiten des Systems erläutert werden. Für weiterführende Informationen, Tipps und Tricks bieten sich die Recherche im Internet und das Studium klassischer Heimcomputerzeitschriften an.[2]

Ergänzend sind in die 5. Auflage die Beschreibungen von 'Action! Toolkit' und 'Action! Runtime Library' eingeflossen.

Das Toolkit enthält eine Sammlung an Routinen und Beispielen, die das Programmieren erleichtern und einiges an Wissen bieten.

Die Run Time Library wird für die Weitergabe eigener Programme an ATARIaner ohne Modul benötigt.

Als Referenz zum Action!-System stellt dieses Buch eine Handreichung zum Erlernen von Action! dar. Für das Entwickeln von Ideen und Programmen bleibt man allerdings auf eigene Kreativität und Experimentierfreudigkeit angewiesen. Insofern ist es immer noch eine spannende Herausforderung, Action! als Programmiersprache zu meistern.

2 Handhabung

Einige kurze Informationen für Anwender, die mit dem ATARI 8-Bit-System nicht so vertraut sind, sowie wichtige Hinweise für ATARIaner, die erste Versuche mit Action! unternehmen wollen.

2.1 ATARI Computer vorbereiten

Der ATARI ist an einen Bildschirm angeschlossen. Der Computer ist mit einem bootfähigen Massenspeichergerät verbunden (Diskettenlaufwerk, Festplatte oder emuliertes Laufwerk). Ein DOS ist vorhanden. Alle Geräte haben eine funktionierende Stromversorgung und sind ausgeschaltet.

- Action!-Modul in den Computer einstecken.
- Bildschirm einschalten.
- BOOT-Laufwerk einschalten (DOS-Disk bzw. Partition auswählen).
- Computer einschalten, DOS wird gebootet.

Danach landet man entweder direkt im Action!-Editor oder im Menü bzw. der Kommandozeile des verwendeten DOS, je nach Konfiguration desselben. Zur weiteren ausführlichen Information bitte das Handbuch des gewählten DOS lesen.

---

2) Anlage L.

2.2 Versionen feststellen

Dieses Handbuch befasst sich mit den aktuellen Versionen von Action!, Action! Toolkit und Action! Runtime Library.

Die vorliegende Version von Action! ermittelt man im Monitor per Eingabe von:

```
?$B000 <RETURN>
```

Das aktuelle Modul liefert daraufhin: `45056,$B000 = 6 $0136 54 310`

Die beiden letzten Stellen der Hexadezimalzahl rechts vom Gleichheitszeichen geben die Version an. Die "36" steht für Version 3.6.

Im aktuellen Toolkit steht in der ersten Zeile der Datei 'ABS.ACT' die Versionsnummer 3. Da es eine reine Textdatei ist, kann man die vorliegende Version von DOS aus prüfen.

Die aktuelle Runtime Library ist die Version 1.4. Da es ebenfalls eine Textdatei ist, kann man die vorliegende Version auch von DOS aus prüfen.

Die Informationen in diesem Handbuch sind auf andere als die oben aufgeführten, aktuellen Versionen nur bedingt, teilweise gar nicht anwendbar. Daher empfehle ich zuerst den Wechsel von ROMs und/oder Dateien auf die aktuellen Versionen.

2.3 Kompatibilität zu DOS und OS

Als Action! 1983 erschien, waren bis dahin ATARI DOS 2.0 und OS/A+ quasi die Standard-DOS, deren Nachfolger ATARI DOS 2.5 und DOS XL generell eine gute Basis für die Verwendung der OSS-Programmiersprachen boten. Seitdem hat sich in der DOS-Welt für den A8 einiges getan und vor allem bei Verwendung von modernen Massenspeichern am A8 gibt es Alternativen.

Grundsätzlich funktioniert Action! unter jedem DOS. Bei intensiver Programmierarbeit kommt allerdings schnell eine umfangreiche Sammlung an Routinen und Programmen zusammen. Von daher empfiehlt sich die Verwendung eines DOS, das mit größeren Medien als Disketten umgehen kann und mit Unterverzeichnissen arbeitet, damit die Routinen strukturiert abgespeichert werden können.

Wer zusätzlich Wert auf eine komfortable Programmierumgebung legt, dem empfehle ich ein DOS, das auf dem SpartaDOS File System aufsetzt. SpartaDOS X V. 4.4x ist das einzig aktuell gehaltene DOS in dieser Kategorie, es wird permanent weiterentwickelt. Der darin vorhandene Tastaturpuffer wurde für Action! angepasst.

Hinsichtlich der vielen OS, die für den A8 verfügbar sind, lässt sich mit einiger Sicherheit sagen, das Action! mit dem Standard-OS des A8 am besten harmoniert. Schwierigkeiten bereiten vor allem OS mit veränderten Routinen für Tastatur und Bildschirm.

3 Das Action!-System

Action! ist eine sogenannte Compilersprache. Das System im Modul besteht aus den fünf Komponenten Monitor, Editor, Sprache, Compiler und Library.

Der Monitor steuert das gesamte Action!-System. Aus ihm heraus werden Editor und Compiler aufgerufen bzw. Zugriffe auf verschiedene Systemoptionen ermöglicht.

Im Editor werden Programme geschrieben bzw. geändert. Es ist ein einfacher Texteditor, der den eingegebenen 'Programmtext' nicht auf Fehler prüft. Deshalb kann der Editor neben der Eingabe von Programmen auch für das Editieren anderer Texte verwendet werden. Außerdem kann man aus dem Editor den im Editorpuffer vorhandenen Text abspeichern oder einen Text in den Editor dazu laden.

Die Sprache dient der Kommunikation mit dem ATARI. Das Programm wird als ATASCII-Text in der Hochsprache Action! geschrieben und anschließend vom Compiler[3] in Maschinensprache übersetzt; erst danach kann es ausgeführt werden. Das erklärt die hohe Geschwindigkeit der Action!-Programme.

Der Compiler prüft das Programm auf Fehler und übersetzt es in Maschinensprache. Nach erfolgreicher Prüfung und Übersetzung kann es ohne weiteren Check gestartet werden. Durch diese Technik wird der Ablauf so enorm beschleunigt.

Die Action! Library (Bibliothek) enthält vordefinierte Programmroutinen, die in eigenen Programmen verwendet werden können. Diese Routinen sind als Bestandteil des Action!-Systems im Modul gespeichert und beinhalten Äquivalente zu den vom BASIC her bekannten Befehlen wie z.B. PLOT, PRINT, DRAWTO usw.

Das 'Action! Runtime Package' macht den User vom Modul unabhängig. Es enthält Library-Routinen, die den im Modul gespeicherten Routinen entsprechen. Zusammen mit eigenen Programmen kompiliert und auf ein Medium gespeichert, ermöglicht die 'RunTime' Programme lauffähig auch ohne Modul weiterzugeben.

Ergänzt wird das Action!-System durch die Programmierhilfe 'Action! Toolkit', deren umfangreiche Routinen und Beispiele im Kapitel VIII erläutert werden.

4 Programmieren in Action!

Am besten macht man sich mit Action! vertraut, indem man mit dem Editor ein kleines Programm schreibt, kompiliert und startet.

Ruft man vom DOS das Modul auf, gelangt man direkt in den Editor und kann sofort mit der Eingabe des Programmtextes beginnen. Tippfehler lassen sich korrigieren, indem man mit den Cursortasten an die entsprechende Stelle 'fährt' und den Fehler einfach überschreibt. Beim Lesen des Kapitels über den Editor gibt es neben den Kommandos noch weitere Feinheiten zu entdecken.

Nun zum Programm. Die Eingabe muss exakt so aussehen:

```
PROC Hallo()
  PrintE("Hallo Leute!")
RETURN
```

Vor dem Kompilieren des Programms eine kurze Betrachtung dessen, was da vor sich geht. Die Ausdrücke 'PROC' und 'RETURN' bilden das Gerüst einer sogenannten Prozedur und werden von der Sprache zur Ablaufsteuerung benötigt. Die Sprache an sich ist in viele Subroutinen gegliedert, die als Prozeduren (PROC) und Funktionen (FUNC) bezeichnet werden. Jede Subroutine führt nur eine vom Autor festgelegte Aufgabe aus.

3) Handoptimierter One-Pass-Compiler.

Das erscheint im ersten Moment sonderbar, erlaubt aber Programme modular aus einzelnen Komponenten zu entwickeln. So kann man sich auf den Teil des Gesamtprogramms konzentrieren, an dem man gerade arbeitet. Dadurch sind die Programme auch sehr viel leichter lesbar als in Spaghetticode geschriebene BASIC-Programme.

Die gerade geschriebene PROC bekommt den Namen "Hallo", weil sie auf dem Bildschirm "Hallo Leute!" ausgibt.

Der Ausdruck 'PrintE' ruft eine Routine auf, die im Modul gespeichert ist. 'PrintE' gibt einen festgelegten Text (String) und an dessen Ende ein <RETURN> aus. Diese Routine ist die einzige Anweisung in der PROC 'Hallo', denn sonst steht ja nichts zwischen 'PROC' und 'RETURN'.

Das Programm befindet sich im Editorpuffer und kann nun kompiliert werden. Dazu aktiviert man mit der Tastenkombination <CTRL><SHIFT><M> den Monitor und ruft von dort per Eingabe von 'C<RETURN>' den Compiler auf, der das Programm auf Fehler überprüft und in Maschinensprache übersetzt.

Der Compiler tut nun seine Arbeit. Wird ein Fehler gefunden, gibt der Compiler eine Fehlermeldung[4] aus und man wird mit einem Hinweis zurück in den Monitor geschickt. Mit 'E<RETURN> gelangt man dann wieder in den Editor und kann den Fehler beheben.

Beachtenswert dabei ist, dass der Cursor im Editor an der Stelle steht, an der vom Compiler der Fehler entdeckt wurde. Somit muss man nicht erst lange nach dem Fehler suchen.

Wird kein Fehler entdeckt, gelangt man auch zurück in den Monitor. Von dort wird das kompilierte Programm durch Eingabe von 'R<RETURN>' gestartet. Auf dem Bildschirm erscheint nun:

```
Hallo Leute!
```

Das war das erste eigene Programm in Action!.

Natürlich kann man bereits vorhandene Programmtexte einladen und ausprobieren, um daran zu lernen. Dabei gilt es allerdings dem DOS bzw. der Konfiguration desselben Rechnung zu tragen. So können DOS-Funktionen oder Befehle unterschiedlich sein, der freie Speicher eventuell nicht ausreichen, zuvor geladene Handler (Treiber) nicht kompatibel zu Action! sein, oder es wird gar Hardware angesprochen, die im eigenen System nicht zur Verfügung steht. Nicht immer geben die Fehlermeldungen sofort den Hinweis auf das Problem. Wenn also ein Programm nicht oder nicht wie gewünscht funktioniert, muss es nicht an Action! liegen.

Hinweis: Kompilierte Action!-Programme werden von einigen DOS beim Laden zweimal ausgeführt. Die Bildschirmausgabe von 'Hallo' sieht dann so aus:

```
Hallo Leute!
Hallo Leute!
```

---

4) Die Fehlercodes sind im Anhang C aufgelistet.

Gelöst wird das Problem[5] durch Einfügen dieser globalen Variable als allererste Zeile in das eigene Programm:

```
BYTE RTS=[$60]
```

Das ergänzte Programm 'Hallo' sieht dann so aus:

```
BYTE RTS=[$60]

PROC Hallo()
  PrintE("Hallo Leute!")
RETURN
```

SpartaDOS X ab Version 4.47 behebt das Problem durch einen entsprechenden Modus zum Laden solcher Binärfiles[6].

5) Weitere Tipps in Anhang K.
6) SpartaDOS X User Guide V. 4.47 - Kommandos CAR und LOAD.

II. Der Action!-Editor

1 Einführung

Der Editor ist der Teil des Systems, in dem Programme eingegeben und geändert werden. Wer schon einmal mit einem Programmeditor gearbeitet hat, wird feststellen, dass der Action!-Editor viel raffinierter als manch anderer ist. Tatsächlich kann er fast schon als Textverarbeitung bezeichnet werden, da er so viele Möglichkeiten bietet.

Trotz dieser Leistungsfähigkeit ist das Arbeiten mit diesem Editor wirklich einfach. Wer nur den ATARI-Editor kennt, wird eine positive Überraschung erleben. Der Editor eignet sich für alles, was es zu editieren gilt. Man kann damit Briefe, Programme in anderen Programmiersprachen oder einfach nur Texte schreiben.

1.1 Festgelegte Schreibweisen und Begriffe

Hochkomma (')

Im Handbuch werden Kommandos und spezielle Zeichen in Hochkommata eingeschlossen.

'<' und '>'

Tastenbezeichnungen werden in '<' und '>' eingeschlossen. <BK SP> ist z.B. die Darstellung der <DELETE BACK SPACE>-Taste. Manche Tasten haben mehr als eine Beschriftung. Dann wird die Bezeichnung verwendet, welche die Editorfunktion am besten beschreibt.

Mehrfach-Tasten-Kommandos

Für einige Editorkommandos müssen mehrere Tasten gleichzeitig gedrückt werden. Die zu drückenden Tasten werden in der betreffenden Reihenfolge ohne Leerzeichen aneinandergereiht. Z.B. bedeutet <SHIFT><DELETE>, dass zuerst <SHIFT> gedrückt und festgehalten werden muss, danach ist <DELETE> zu drücken.

Die Informationszeile

Damit wird die invers dargestellte Zeile am unteren Bildrand bezeichnet. Normalerweise steht dort:

```
Action! (c) 1983 ACS
```

Manchmal greift der Editor auf diese Infozeile zu, um dort Informationen, Fragen oder Fehlermeldungen auszugeben. Wird der Editor im Zweifenstermodus[7] benutzt, trennt die Infozeile die beiden Fenster.

Letzte Eingabe des Anwenders

Bei einigen Kommandos erscheint in der Infozeile die Eingabe als Voreinstellung, die bei der letzten Anwendung des Kommandos vom Anwender dort eingegeben wurde. Sofern die angezeigte Information korrekt ist, braucht man nur <RETURN> zu drücken. Oder man ändert die Vorgabe teilweise oder vollständig und drückt dann <RETURN>.

1.2 Konzeption des Editors

Den Bildschirm kann man sich als Fenster vorstellen. Die sichtbare Größe im Editor beträgt 24 Zeilen zu je 38 Zeichen im Standardmodus des ATARIs. Das erschient ziemlich begrenzt und wäre es wohl auch, könnte man dieses Fenster nicht bewegen.

7) Kapitel II, Abschnitt 2.6.2.

Editorfenster
Das Fenster kann auf und ab sowie links und rechts bewegt werden, um den gesamten Text zu betrachten. Es zeigt also immer einen Bildausschnitt einer größeren Textfläche.

Doch der Editor kann noch mehr! Geht eine Programmzeile über den Bildschirmrand hinaus, lässt sich der Cursor in der Zeile bis zum Ende fahren. Nur diese Zeile bewegt sich, ohne das übrige Fenster zu verändern. Verlässt man die Zeile, wird sie wieder linksbündig dargestellt.

Und der Bildschirm kann in zwei Fenster aufgeteilt werden, die unabhängig voneinander sind. Dadurch lassen sich zwei verschiedene Programme oder unterschiedliche Teile des gleichen Programms gleichzeitig bearbeiten.

Textzeilen
Der Editor ist auf gute Lesbarkeit ausgelegt und bietet eine maximale Zeilenlänge von 240 Zeichen. Das wiederum garantiert problemloses Schreiben von strukturierten Programmen, ohne sich Gedanken über die Zeilenlänge machen zu müssen. Auch Leerzeilen zur optischen Trennung einzelner Programmteile sind erlaubt. Dadurch lassen sich Texte optimal gestalten. Das Zeilenende wird bei erreichen mit dem vom BASIC her bekannten Warnton angezeigt.

Ist eine Textzeile länger als 38 Zeichen, wird das Zeichen am linken bzw. rechten Rand zur Erinnerung invers dargestellt.

Suchen und Ersetzen
Der Editor kann nach einer beliebigen Zeichenkette (String) suchen. Er setzt den Cursor dann im Text an die Position, an welcher der gesuchte String zuerst vorkommt. Gefunden wird nur die exakte Schreibweise des Begriffs in Groß-/Kleinbuchstaben).

Als Erweiterung der Suchfunktion ist es zudem möglich, den gefundenen String durch einen anderen String zu ersetzen.

Gefunden wird nur die exakte Schreibweise des Begriffs in Groß-/Kleinbuchstaben.

Verschieben von Textblöcken
Der zu verschiebende Textblock wird zunächst vom Bildschirm entfernt und im Kopierpuffer zwischengespeichert. Per Tastenkombination kann er dann an der jeweiligen Cursorposition eingefügt werden. Man kann den Textblock zuerst an der ursprünglichen Stelle einfügen und danach an jeder beliebigen anderen. Auf diese Weise wird ein Textblock nicht nur verschoben, sondern sogar mehrfach kopiert.

Cursorsteuerung
Der Cursor wird nicht nur durch die Cursortasten gesteuert. Er kann auch mit dem Kommando 'FIND' (finden) oder durch setzen von Markierungen (tags) bewegt werden.

Markierungen (tags)
Jede Position innerhalb des Textes kann mit einer unsichtbaren Markierung versehen werden. Per Kommando kann der Cursor dann unmittelbar dahin springen. Die Anzahl der möglichen Markierungen ist durch die Anzahl der verfügbaren Zeichen beschränkt, da jede Position mit einem anderen Zeichen gekennzeichnet werden muss.

2 Die Editorkommandos

Die Überschriften geben die jeweilige Funktion an. Das dient der besseren Übersicht beim Lesen und Nachschlagen. Jedes Editorkommando[8] kann mit <ESC> sofort und sauber abgebrochen werden. <RESET> sollte nur im äußersten Notfall benutzt werden.

2.1 Editor starten

Beim ersten Start des Action!-Systems gelangt man automatisch in den Editor. Verlässt man Action! und geht ins DOS, so landet man später beim erneuten Aufruf von Action! im Monitor und erreicht von dort mit <E><RETURN> den Editor.

Löscht man unter SpartaDOS X eine ggf. vorhandene Datei 'CAR.SAV', wird das Modul ebenfalls wie beim Kaltstart initialisiert und man landet direkt im Editor. Ab Sparta-DOS X Version 4.43 steht dafür der Kommandoparameter 'CAR /I' zur Verfügung[9].

2.2 Editor verlassen

Zum Verlassen des Editors gibt es nur einen Weg: <CTRL><SHIFT><M>.

Man gelangt dadurch in den Monitor, von wo alle anderen Komponenten des Action!-Systems erreichbar sind bzw. das System nach DOS verlassen werden kann.

2.3 Texteingabe

Text wird im Editor einfach eingetippt. Erreicht man das Ende der festgelegten Zeilenlänge, ertönt der ATARI-eigene Warnton, auch als Summer ('Bell') bezeichnet.

Für die Eingabe eines Sonderzeichens (control character) muss zuvor <ESC> gedrückt werden. Dadurch erkennt der Editor das Zeichen als Text und interpretiert es nicht als Editorkommando.

Das Ändern von bereits eingegebenem Text ist auf zwei Arten möglich. Entweder durch Überschreiben (replace) oder durch Einfügen (insert).

Beim Überschreiben wird der alte Text einfach durch den neuen ersetzt.

Beim Einfügen wird dagegen der neue Text an der Cursorposition eingefügt. Zwischen diesen beiden Modi wird mit <SHIFT><CTRL><I> umgeschaltet. Der jeweils ausgewählte Modus wird in der Infozeile angezeigt.

Beim ersten Start des Editors ist der Modus Überschreiben aktiv.

Soll der gesamte im Fenster befindliche Text gelöscht werden, setzt man den Cursor in das entsprechende Fenster und drückt <SHIFT><CLEAR>. Der gesamte Text, nicht nur der sichtbare Ausschnitt, wird dann unwiederbringlich gelöscht.

2.3.1 Ein-/Ausgabe von Textdateien

Der Editor ermöglicht die Kommunikation zum Laden, Speichern und Drucken von Programmen mit angeschlossenen Peripheriegeräten wie Diskettenlaufwerk, Kassettenrekorder, Drucker usw. Der Modus entspricht dem von LIST und ENTER im ATARI BASIC.

---

8) Anhang D: Übersicht der Editorkommandos.
9) SpartaDOS X Manual bzw. ab V. 4.47 User Guide auf http://sdx.atari8.info/

Um ein im Editor befindliches Programm abzuspeichern, wird zuerst der Cursor im entsprechenden Fenster positioniert (bei nur einem Fenster nicht nötig) und dann das Kommando <CTRL><SHIFT><W> eingegeben. Daraufhin erscheint in der Infozeile

```
Write? (Schreiben/Ausgeben)
```

Nun gibt man das Zielgerät und ggf. den Namen des Programms ein. Bei DOS mit Unterverzeichnissen kann auch der Pfad zum Zielverzeichnis angegeben werden.

Wer ohne DOS arbeitet, kommt mit ‘C:’, ‘P:’ etc. für das Zielgerät aus (C: für Kassettenlaufwerk, P: für Drucker).

Das Laden eines Programms ist genauso einfach. Der Cursor wird im Textfenster an die Stelle positioniert, an die der Programmtext geladen werden soll. Wird dann das Kommando <CTRL><SHIFT><R> eingegeben, erscheint in der Infozeile die Frage:

```
Read? (Einlesen)
```

Gibt man nun den Namen des zu ladenden Programms ein, wird es vom aktiven Laufwerk aus dem aktuellen Verzeichnis geladen. Wie beim Speichern können im Pfad Geräte und Unterverzeichnisse angegeben werden. Da der Text wie beim BASIC-Befehl ENTER eingelesen wird, wird nichts gelöscht. Soll der im Editor vorhandene Text nicht weiter verwendet werden, bitte zuvor mit <SHIFT><CLEAR> löschen.

Das aktuelle Inhaltsverzeichnis eines Laufwerks wird eingelesen, wenn die Frage "Read?" in der Infozeile mit '?:*.*' beantwortet wird. Mit '?1:*.*' wird Laufwerk #1 angesprochen. Für den Zugriff auf ein anderes Laufwerk die 1 durch die jeweilige Nummer ersetzen. So kann man schnell herausfinden, was auf dem Laufwerk gespeichert ist. Zusätzlich kann man den Pfad zum gewünschten Verzeichnis angeben. Unter SpartaDOS X ist die Laufwerkskennung 'DA:' bis 'DO:' zulässig. Die von Action! akzeptierte maximale Anzahl von Zeichen für den Pfad beträgt dabei nur 33 Zeichen plus EOL[10].

2.3.2 Zeilenlänge festlegen

Wie schon unter 2.3 angedeutet, kann die Zeilenlänge vom Anwender festgelegt werden. Das erfolgt im Optionsmenü des Compilers[11].

2.4 Cursor bewegen

Cursor hoch	<CTRL><↑>	oder	<F1>
Cursor 'runter	<CTRL><↓>	oder	<F2>
Cursor links	<CTRL><←>	oder	<F3>
Cursor rechts	<CTRL><→>	oder	<F4>

Diese Kommandos versteht auch der ATARI-BASIC-Editor. Der Action!-Editor verfügt noch über weitere Cursorkommandos:

Cursor an Zeilenanfang	<CTRL><SHIFT><**<**>
Cursor an Zeilenende	<CTRL><SHIFT><**>**>

10) Beispiel für Laufwerk #5: DE:>ACTION>GAMES>POTTER.ACT
11) Kapitel III, Abschnitt 2.5.

Diese beiden Kommandos bringen den Cursor an den tatsächlichen Anfang bzw. das tatsächliche Ende der Zeile, auch wenn es im Fenster nicht sichtbar ist. Wird der Cursor danach in eine andere Zeile bewegt, springt die Zeile in die alte Position zurück.

Cursor in erste Textzeile <CTRL><SHIFT><H>
Cursor in letzte Textzeile <CTRL><SHIFT><E>

Dabei wird der Cursor jeweils auf die linke Randposition des aktuellen Fensters gesetzt. Wurde das Fenster mit <CTRL><SHIFT><]> vom linken Rand nach rechts verschoben, so steht der Cursor nicht am Zeilenanfang.

2.4.1 Tabulatoren

Der Cursor wird durch Drücken von <TAB> zur nächsten Position bewegt.

Zum Einrichten eines neuen Tabstopps den Cursor an die jeweilige Position fahren und <SHIFT><TAB> drücken.

Zum Löschen eines Tabstopps den Cursor entsprechend positionieren und <CTRL><TAB> drücken.

2.4.2 Auffinden von Textstellen

Der Editor bietet über das Kommando <CTRL><SHIFT><F> eine Suchoption für Strings von maximal 32 Zeichen Länge. In der Infozeile wird darauf die Frage 'Find?' (Finde?) ausgegeben. Wurde das FIND-Kommando schon zuvor angewendet, wird der zuletzt verwendete Suchbegriff angezeigt. Soll dieser String erneut gesucht werden, braucht nur noch <RETURN> gedrückt zu werden. Soll ein anderer String gefunden werden, gibt man ihn ein und drückt <RETURN>. Der alte String verschwindet beim Tippen automatisch.

Beim ersten Mal ist natürlich noch kein String vorhanden. Also String eingeben und <RETURN> drücken.

Dieses Kommando wird immer von der momentanen Cursorposition an zum Textende hin ausgeführt. Sobald die nächste Position des gesuchten Strings erreicht ist, setzt der Editor den Cursor auf das erste Zeichen des gefundenen Strings. Wird der String nicht gefunden, so gibt der Editor in der Infozeile die Meldung 'not found' (nicht gefunden) aus. Gefunden wird nur die exakte Schreibweise des Begriffs in Groß-/Kleinbuchstaben.

2.5 Text korrigieren

In diesem Abschnitt wird erklärt, wie im Editor Text geändert und gelöscht wird bzw. wie bereits gelöschter Text eventuell zurückgeholt werden kann.

2.5.1 Ein Zeichen löschen

<CTRL><DELETE> löscht das Zeichen unter dem Cursor. Dadurch rückt das rechts vom Cursor stehende Zeichen nach links und füllt die entstandene Lücke.

<BK SP> löscht das links vom Cursor stehende Zeichen. Im Modus Überschreiben wird das Zeichen links vom Cursor durch ein Leerzeichen ersetzt. Im Modus Einfügen dagegen wird das Zeichen links vom Cursor gelöscht und alle folgenden Zeichen werden herangerückt.

2.5.2 Einfügen - Überschreiben

Wie schon unter 2.3 erläutert, gibt es zwei verschiedene Modi für die Texteingabe: Überschreiben und Einfügen. Nach dem Start des Editors befindet man sich im Modus Überschreiben. Umschalten erfolgt durch <CTRL><SHIFT><I>.

Einige Editorkommandos sind vom gewählten Modus abhängig, da sie unterschiedliche, eben vom Modus abhängige Funktionen ausführen.

Ein Leerzeichen wird an der Cursorposition mit <CTRL><INSERT> eingefügt. Der rechts vom Cursor stehende Text wird um ein Zeichen nach rechts gerückt und an der Cursorposition wird ein Leerzeichen in den Text eingefügt. Im Modus Einfügen muss dazu nur die Leertaste gedrückt werden.

2.5.3 Zeile löschen

Mit <SHIFT><DELETE> wird die Zeile gelöscht, in welcher der Cursor derzeit steht. Die nachfolgenden Zeilen rücken darauf um eine Zeile nach oben.

2.5.4 Zeile einfügen

Mit <SHIFT><INSERT> wird eine Zeile eingefügt. Die nachfolgenden Zeilen werden um eine Zeile nach unten verschoben, um Platz für die neue Zeile zu schaffen.

2.5.5 Zeilen trennen und zusammenfügen

Soll eine Zeile in zwei Zeilen aufgetrennt werden, muss zuerst der Cursor auf das Zeichen positioniert werden, das als Erstes in der zweiten Zeile stehen soll. Dann drückt man <CTRL><SHIFT><RETURN>.

Im Modus Einfügen braucht man dazu nur den Cursor zu positionieren und <RETURN> zu drücken. Die nachfolgenden Zeilen werden um eine Zeile nach unten verschoben, um Freiraum zu bekommen.

Zwei Zeilen werden mit <CTRL><SHIFT><BK SP> zu einer Zeile zusammengefügt. Dazu positioniert man den Cursor auf das erste Zeichen der zweiten Zeile und drückt obige Tastenkombination. Die nachfolgenden Zeilen werden nach oben gerückt.

2.5.6 Text ersetzen

Im Editor wird mit <CTRL><SHIFT><S> durch "neuen Text" zu suchender "alter Text" ersetzt. Es erscheint in der Infozeile die Frage:

```
Substitute? (Ersatztext? = neuer, einzufügender Text?)
```

Zuerst wird der neue Text eingegeben, danach der alte Text. Der Editor sucht ab der derzeitigen Cursorposition nach dem alten Text und ersetzt ihn dann durch den neuen.

War das Kommando zuvor schon einmal benutzt worden, wird der zuletzt verwendete Text angezeigt. Soll dieser erneut zum Ersetzen verwendet werden, braucht man nur noch <RETURN> zu drücken. Andernfalls gibt man den neuen Text ein und drückt dann <RETURN>. Danach erscheint in der Infozeile:

```
for? (für? = alter, zu ersetzender Text)
```

War diese Funktion zuvor schon einmal ausgeführt worden, erscheint jetzt in der Infozeile der zuletzt ersetzte alte String. Soll er wieder ersetzt werden, braucht man nur <RETURN> zu drücken. Andernfalls muss der jetzt zu ersetzende String eingegeben und <RETURN> gedrückt werden.

Wird diese Funktion das erste Mal ausgeführt, ist natürlich noch kein alter String vorhanden.

Nach <RETURN> sucht der Editor dann den alten String und ersetzt ihn durch den neuen. Findet der Editor nichts und kann deshalb die Funktion nicht ausführen, gibt er in der Infozeile die Nachricht aus:

```
not found (nicht gefunden)
```

Soll der alte String mehrfach gesucht und ersetzt werden, muss mehrmals hintereinander <CTRL><SHIFT><S> gedrückt werden, ohne eine Änderung der Strings vorzunehmen. Auf diese Weise lässt sich die Ersetzen-Funktion sehr schnell mehrmals ausführen, ohne die Fragen "Substitute?" und "for?" jedes Mal beantwortet zu müssen.

Mit dieser Funktion kann auch nur Text gelöscht werden, indem man für den neuen String nichts eingibt. Dadurch wird der alte String durch nichts ersetzt, also gelöscht.

2.5.7 Zeilenänderung rückgängig machen

Der Editor erlaubt mit <CTRL><SHIFT><U> den vorherigen Zustand einer geänderten Zeile wieder herzustellen. Auf diese Weise kann ein Fehler schnell korrigiert werden. Dieses Kommando funktioniert nur solange, wie man die Zeile nicht verlässt. Wurde einmal <RETURN> gedrückt, kann der Zeilentext nicht mehr zurückgesetzt werden.

Eine versehentlich gelöschte Zeile kann mit <CTRL><SHIFT><P> zurückgeholt werden[12]. Waren in der veränderten bzw. gelöschten Zeile Markierungen (tags) gesetzt, so sind diese leider verloren.

2.6 Fenster

Der Editortext ist in einem Fenster zu sehen. Nachfolgend wird erläutert, wie mit den Editorkommandos Fenster eingerichtet, verändert oder gelöscht werden. Mit dem angesprochenen Fenster ist jeweils das derzeit verwendete Fenster gemeint.

2.6.1 Fenster bewegen

Mit dem Cursor wird das Fenster zeilenweise auf und ab bewegt. Diese Art den Text "weiterzukurbeln" ist für lange Programme weniger geeignet. Besser geht das durch "seitenweises Blättern".

Mit <CTRL><SHIFT><↑> blättert man nach oben. Damit man dabei die Übersicht nicht verliert, wird jeweils die oberste Zeile auf der nächsten Seite als unterste Zeile angezeigt.

In die Gegenrichtung geht es mit <CTRL><SHIFT><↓>.
Zur Übersicht wird die unterste Zeile dann als oberste Zeile auf der nächsten Seite angezeigt.

12) Mehr dazu in Kapitel II, Abschnitt 2.7.

Das Fenster lässt sich auch horizontal bewegen. Überschreitet dabei eine Zeile die festgelegte Bildschirmbreite[13], so wird das am entsprechenden Rand befindliche Zeichen als Markierung invers dargestellt.

Mit <CTRL> <SHIFT><]> wird das Fenster nach rechts, mit <CTRL><SHIFT><[> nach links bewegt.

Befindet sich das Fenster in der äußersten linken bzw. rechten Position, kann es selbstverständlich nur noch in die jeweilige Gegenrichtung bewegt werden.

2.6.2 Zweites Fenster einrichten

Nach dem Start des Editors sieht man nur ein Fenster. Ein zweites Fenster wird mit <CTRL><SHIFT><2> eingerichtet. Der Bildschirm ist nun durch die Infozeile in zwei Fenster aufgeteilt. Oberhalb der Infozeile befindet sich Fenster #1, unterhalb Fenster #2. Die beiden Fenster sind voneinander unabhängig und erlauben die Arbeit an zwei verschiedenen Programmen oder Texten.

Die Größe des Fensters #1 kann über das Options-Menü verändert werden[14]. Fenster #2 nimmt den jeweiligen Rest des Bildschirms ein.

2.6.3 Fenster anwählen

Mit <CTRL><SHIFT><2> wechselt man von Fenster #1 zu Fenster #2. War Fenster #2 noch nicht vorhanden, wird es hierdurch geöffnet und der Cursor darin aktiviert. Mit <CTRL><SHIFT><1> geht es zurück zu Fenster #1[15].

2.6.4 Fenster löschen

Will man den gesamten Text aus einem Fenster löschen, aktiviert man das entsprechende Fenster und drückt dann <SHIFT><CLEAR>. Da dies eine ultimative Funktion ist, erfolgt in der Infozeile eine Sicherheitsabfrage:

```
CLEAR?  (Löschen?)
```

Diese kann mit "Y" für Ja bzw. "N" für Nein beantwortet werden. Wurde der Inhalt des Fensters geändert aber noch nicht abgespeichert, erscheint in der Infozeile die Frage:

```
Not saved, Delete?  (Noch nicht abgespeichert, löschen?)
```

Auf diese Weise wird verhindert, dass ein Programm versehentlich zerstört wird. Dieses Kommando löscht nicht nur den im Fenster sichtbaren Text, sondern den ganzen, in diesem Teil des Editors gespeicherten Text.

2.6.5 Fenster schließen

Zum Schließen eines Fensters (es verschwindet dann vom Bildschirm) wird das Fenster zuerst mit <CTRL><SHIFT><1> oder <2> aktiviert und dann die Tastenkombination <CTRL><SHIFT><D> gedrückt. In der Infozeile erscheint die Sicherheitsabfrage:

```
Delete window?  (Fenster schließen?)
```

---

13) Mehr dazu in Kapitel III., Abschnitt 2.5.
14) Mehr dazu in Kapitel III., Abschnitt 2.5.
15) Kompilieren aus dem aktiven Fenster, Kapitel V, Abschnitt 1.

Als Antwort ist nur "Y" oder "N" zulässig. Wurde der Inhalt des Fensters geändert aber noch nicht abgespeichert, erscheint die Frage:

```
Not saved, Delete?  (Noch nicht abgespeichert, löschen?)
```

So wird verhindert, dass ein Programm versehentlich zerstört wird.

Mit dem Schließen verschwindet ein Fenster vom Bildschirm. Durch Schließen von Fenster #1 wird Fenster #2 zu Fenster #1.

2.7 Textblöcke verschieben und kopieren

Der Editor verfügt über einen Kopierpuffer, der das Verschieben und Kopieren von Textblöcken ermöglicht. Jede mit <SHIFT><DELETE> gelöschte Zeile wird im Kopierpuffer zwischengespeichert. Mit der Tastenkombination <CTRL><SHIFT><P> kann diese Zeile an anderer Stelle eingefügt werden. Durch erneutes Anwenden des Kommandos <SHIFT><DELETE> wird der bisherige Inhalt des Kopierpuffers gelöscht und durch die zuletzt gelöschte Zeile ersetzt.

Davon gibt es aber eine wichtige Ausnahme: Wird das Kommando mehrmals hintereinander ausgeführt, ohne dass dazwischen andere Kommandos oder Texteingaben erfolgten, wird der Kopierpuffer nicht gelöscht. Stattdessen werden alle auf diese Weise gelöschten Zeilen hintereinander in den Kopierpuffer übertragen. Das erst ermöglicht Blockoperationen.

Beim nächsten <CTRL><SHIFT><P> wird dann der gesamte Block aus dem Kopierpuffer in den Text eingefügt. Soviel als erster Überblick, nun zu den Details.

Soll ein Textblock verschoben werden, positioniert man den Cursor in die erste Zeile des Blocks und drückt solange <SHIFT><DELETE> bis der ganze Block gelöscht ist. Nun bewegt man den Cursor in die Zeile, ab welcher der Block eingefügt werden soll, drückt <CTRL><SHIFT><P> und der Block wird eingefügt.

Zum Kopieren eines Blocks verwendet man die gleiche Methode. Nur wird der gelöschte Block zuerst an die Stelle kopiert, von der er "weggelesen" wurde. Anschließend kopiert man ihn an die andere gewünschte Position. Da durch das Einfügen der Kopierpuffer nicht gelöscht wird, lassen sich beliebig viele Kopien erzeugen.

2.8 Markierungen (tags)

Mit Markierungen kann jede beliebige Textstelle markiert werden. Mit <CTRL> <SHIFT>T wird eine Markierung an der aktuellen Cursorposition gesetzt. In der Infozeile erscheint die Aufforderung:

```
tag id:  (Markierungszeichen:)
```

Nun das Markierungszeichen für diese Stelle eingeben und <RETURN> drücken. Wird ein bereits verwendetes Markierungszeichen benutzt, wird es an der bisherigen Position gelöscht und an der jetzt neuen Position gespeichert.

Eine Markierung erreicht man über <CTRL><SHIFT>G. Darauf erscheint in der Infozeile:

```
tag id:
```

Jetzt wird das Markierungszeichen eingegeben, zu dem gesprungen werden soll. Ist das Zeichen vorhanden, geht der Cursor an die entsprechende Stelle und zeigt den um die Markierung vorhandenen Text an. Gibt es das Zeichen nicht, wird in der Infozeile

```
tag not set  (Markierung nicht gesetzt)
```

ausgegeben. Dann ist keine Markierung mit diesem Zeichen belegt.

Wichtig: Die Änderung einer Zeile, in der eine Markierung gesetzt ist, bewirkt das Löschen der Markierung!

3 Vergleich zwischen Action!- und ATARI-Editor

Gemeinsamkeiten und Unterschiede werden in diesem Abschnitt herausgearbeitet, um die Leistungsfähigkeit des Action!-Editors zu erläutern.

3.1 Identische Kommandos

<SHIFT>
In Verbindung mit den Zeichentasten werden Großbuchstaben oder andere Zeichen erzeugt oder Kommandos gegeben.

<CTRL>
In Verbindung mit einer oder mehreren Tasten werden Kommandos oder Spezialzeichen (control characters) erzeugt.

<ATARI>
Die ATARI- oder Invers-Taste schaltet auf inverse Zeichendarstellung um. Nochmaliges Drücken schaltet in den normalen Zustand zurück.

<ESC>
Ermöglicht die Eingabe von Steuerzeichen (control characters) als Text.

<LOWR/CAPS>
Schaltet von Groß- auf Kleinbuchstaben um.

<SHIFT><CAPS>
Schaltet von Klein- auf Großbuchstaben um.

<SHIFT><INSERT>
Fügt an der Cursorposition eine Leerzeile ein. Nachfolgende Zeilen werden nach unten verschoben.

<CTRL><INSERT>
Fügt an der Cursorposition ein Leerzeichen ein.

<CTRL><↑> oder <F1>
Cursor nach oben - stoppt aber in der obersten Textzeile.

<CTRL><↓> oder <F2>
Cursor nach unten - stoppt aber in der untersten Textzeile.

<TAB>
Cursor zur nächsten Tabulatorposition.

<SHIFT><SET TAB>
An der Cursorposition einen neuen Tab setzen.

<CTRL><CLR TAB>
An der Cursorposition den Tab löschen.

3.2 Abweichende Kommandos

<BREAK>
Im Action!-Editor nicht belegt.

<SHIFT><CLEAR>
Löscht das Programm im aktiven Fenster.

<RETURN>
Im Modus Überschreiben springt der Cursor in die nächste Zeile. Im Modus Einfügen wird ein <RETURN> in den Text eingefügt.

<SHIFT><DELETE>
Löscht die Zeile, in der sich der Cursor befindet. Nachfolgende Zeilen werden hoch geholt. Kann wiederholt ausgeführt werden. Gelöschte Zeilen befinden sich im Kopierpuffer und können mit <CTRL><SHIFT>P wieder eingefügt werden.

<BK SP>
Im Modus Überschreiben wird das Zeichen links vom Cursor gelöscht. Im Modus Einfügen wird zusätzlich der Rest der Zeile herangerückt.

<CTRL><←> oder <F3>
Cursor nach links - stoppt am Anfang der Zeile. Erreicht der Cursor den Rand des Fensters, bleibt er stehen und die aktive Zeile wird durch das Fenster verschoben.

<CTRL><→> oder <F4>
Cursor nach rechts - stoppt am Ende der Zeile. Erreicht der Cursor den Rand des Fensters, bleibt er stehen und die aktive Zeile wird durch das Fenster verschoben.

3.3 Reine Action!-Kommandos

<CTRL><SHIFT><D>
Schließt das aktive Fenster. Der Inhalt ist verloren, wenn er nicht gespeichert wurde.

<CTRL><SHIFT><E>[16]
Bringt den Cursor in die letzte Zeile.

<CTRL><SHIFT><F>
Findet einen angegebenen Suchstring innerhalb des Textes.

<CTRL><SHIFT><G>
Springt zu einer Markierung innerhalb des Textes.

---

16) Kapitel II, Abschnitt 2.4 zu <E> und <H>.

<CTRL><SHIFT><H>
Bringt den Cursor in die erste Zeile.

<CTRL><SHIFT><I>
Umschalten zwischen den Modi Überschreiben und Einfügen. Dieses Kommando beeinflusst die Funktion von <BK SP> und <RETURN>.

<CTRL><SHIFT><M>
Führt in den Action!-Monitor.

<CTRL><SHIFT><P>
Fügt an der Cursorposition den Inhalt des Kopierpuffers ein.

<CTRL><SHIFT><R>
Laden eines Programms von einem Speichergerät. Die Parameter eines ggf. verwendeten DOS müssen dabei berücksichtigt werden.

<CTRL><SHIFT><S>
Suchen und Ersetzen einer maximal 32 Zeichen langen Zeichenkette.

<CTRL><SHIFT><T>
An der Cursorposition wird eine Markierung gesetzt.

<CTRL><SHIFT><U>
Schreibt eine veränderte oder gelöschte Zeile im Originalzustand zurück. Funktioniert aber nur, solange die Zeile nicht mit <SHIFT><DELETE> gelöscht oder verlassen wurde.

<CTRL><SHIFT><W>
Speichert ein Programm auf einem Speichergerät ab. Die Parameter des ggf. verwendeten DOS müssen dabei berücksichtigt werden.

<CTRL><SHIFT><]>
Bewegt das ganze Fenster um eine Spalte nach rechts.

<CTRL><SHIFT><[>
Fenster um eine Spalte nach links.

<CTRL><SHIFT><↑>
Eine "Seite" nach oben "blättern". Zur besseren Übersicht wird dabei die oberste Zeile zur untersten Zeile auf der neuen Seite.

<CTRL><SHIFT><↓>
Eine "Seite" nach unten "blättern". Zur besseren Übersicht wird dabei die unterste Zeile zur obersten Zeile auf der neuen Seite.

<CTRL><SHIFT><1>
Vom Fenster #2 ins Fenster #1 wechseln.

<CTRL><SHIFT><2>
Vom Fenster #1 ins Fenster #2 wechseln. War das Fenster #2 noch nicht eröffnet, so geschieht dies hiermit.

<CTRL><SHIFT>< > >
Cursor springt ans Ende der maximal 240 Zeichen langen Zeile.

<CTRL><SHIFT>< < >
Cursor springt an den Anfang der maximal 240 Zeichen langen Zeile.

<CTRL><SHIFT><BK SP>
Am Anfang einer Zeile ausgeführt, bewirkt das Kommando, dass diese Zeile an die vorhergehende Zeile angefügt wird.

<CTRL><SHIFT><RETURN>
Fügt im Text ein <RETURN> ein. Eine Zeile wird dadurch in zwei Zeilen aufgetrennt.

4 Technische Anmerkungen
Wichtige Hinweise zum Arbeiten mit dem Editor.

4.1 Text von anderen Editoren
Der Editor kann nur Text verarbeiten, der am Zeilenende <RETURN>-Zeichen (EOL) aufweist und maximal 240 Zeichen lange Zeilen hat. Am besten orientiert man sich für die Zeilenlänge[17] am verwendeten Drucker.

4.2 Tastaturabfrage
Bei Eingabe von Kommandos in der Infozeile werden nur <ESC>, <BK SP> und <CLEAR> erkannt. Für die Texteingabe sind alle Textzeichen und Kommandotasten zugelassen.

4.3 "Out of Memory"-Fehler
Die Meldung "Zu wenig Speicher" wird schon mal dadurch verursacht, dass beim Editieren zu viel eingefügt, ersetzt oder ein zu langes Programm geladen wurde.

Tritt dieser Fehler auf, sofort das im Editor befindliche Programm auf einem Peripheriegerät abspeichern und Action! erneut starten über das 'BOOT'-Kommando im Monitor durch Eingabe von 'B<RETURN>'. Danach kann man wieder in den Editor gehen, das Programm einladen und weiter bearbeiten.

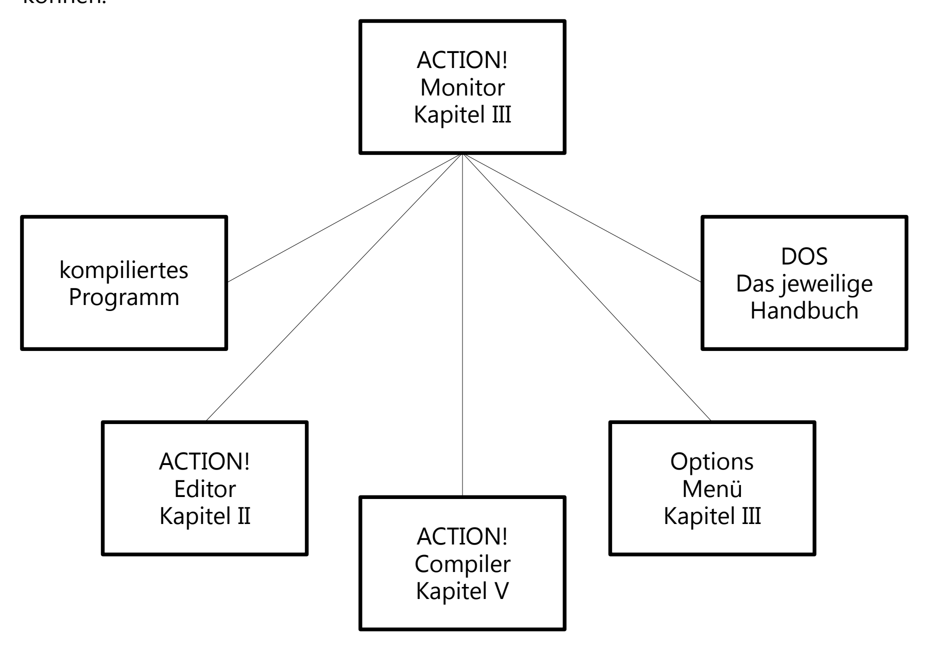
Sollte ein Action!-Programm zu groß geworden sein, wird es in mehrere Einzelprogramme aufgeteilt, die dann einzeln ohne Probleme bearbeitet werden können. Zum Kompilieren werden sie dann vom Speichergerät geladen und zusammengefügt[18].

---
17) Kapitel III, Abschnitt 2.5.
18) Kapitel V, Abschnitt 2.4.

III. Der Action!-Monitor

1 Einführung

Der Action!-Monitor ist das Kontrollzentrum des Systems. Er fungiert als Bindeglied zwischen allen Funktionseinheiten. Das Diagramm veranschaulicht die zentrale Aufgabe des Monitors und gibt die Kapitel an, in denen die Details nachgeschlagen werden können.



Den Monitor erkennt man an der invers dargestellten Titelzeile, in der das Zeichen '>' und der Cursor stehen.

1.1 Begriffsdefinitionen

Begriff	**definiert unter**
<Adresse>	Kapitel IV. - Abschnitt 2.1
<Compilerkonstante>	Kapitel IV. - Abschnitt 3.2
<filespec>	hiernach
<Kennung>	Kapitel IV. - Abschnitt 2.1
<Anweisung>	Kapitel IV. - Abschnitt 5
<Wert>	Kapitel IV. - Abschnitt 3.4.1

Als 'filespec' wird die Angabe des angesprochenen Gerätes und im Falle einer Diskettenstation bzw. Festplatte zusätzlich der Pfad- und/oder Dateiname bezeichnet. Die ATARI-typischen Angaben lauten:

C:	bei Cassette
P:	bei Drucker
D#:PROGNAME.EXT	bei Diskettenstation oder Festplatte.
D#:PFAD>PROGNAME.EXT	bei Diskettenstation oder Festplatte und DOS mit Unterverzeichnissen.

1.2 Konzept des Monitors

Der Monitor stellt zwei Hauptfunktionen bereit:

- die Kommandozeile
- das Informationsfeld.

Der Monitor wird über die invers dargestellte Kommandozeile am oberen Bildschirmrand bedient. Darin steht linksbündig '>' und danach der blinkende Cursor. Als Monitorkommando wird das erste eingegebene Zeichen erkannt. Es wird also mit 'E', 'Elise' und 'Egon' immer der Editor aufgerufen. Die verschiedenen Kommandos des Monitors werden im nachfolgenden Abschnitt 2 erklärt.

Der Bildschirmteil unterhalb der Kommandozeile wird als Informationsfeld bezeichnet. Es handelt sich dabei um ein Mehrzweckfeld. Wird ein Programm gestartet, so gibt der Monitor dort die Ergebnisse aus. Das Feld kann auch benutzt werden, um mit der 'Trace'-Funktion[19] den Programmablauf schrittweise zu verfolgen. Finden entweder das Betriebssystem oder der Compiler einen Fehler, so werden die Fehlernummer und der Programmteil, in dem sich der Fehler befindet, im Infofeld ausgegeben.

Der Monitor ist die Kommandozentrale des Systems. Von dort werden alle anderen Systembereiche aufgerufen. Eine Vorstellung von den Beziehungen vermittelt die Grafik am Anfang dieses Kapitels. Vom Monitor werden kompilierte Programme gestartet, der Editor oder Compiler aufrufen sowie der Sprung zurück ins DOS ausgeführt.

2 Monitorkommandos

Die Kommandos bestehen aus einem Zeichen und werden mit <RETURN> ausgelöst.

2.1 BOOT - Neustart von Action!

Manchmal muss Action! aus dem Monitor heraus erneut gestartet werden, weil ein verhängnisvoller Fehler aufgetreten ist. Die Eingabe dazu lautet: 'B<RETURN>'.

Dabei geht leider der im Editor vorhandene Text verloren. Bereits fertig kompilierte Programme und die dazugehörigen Variablen werden gelöscht.

2.2 COMPILE - Programme kompilieren

Ein Action!-Programm muss erst durch den Compiler verarbeitet werden, bevor es vom Monitor aus gestartet werden kann. Er wird vom Monitor aus mit 'C<RETURN>' aufgerufen. Mit dem Kommando 'C“filespec“<RETURN> wird der Text eines Programms direkt vom Speichergerät geladen und kompiliert. Die Anführungszeichen an Anfang und Ende nicht vergessen! Der Text wird dabei nicht in den Editor geladen. Ohne Angabe einer "filespec" wird der im aktiven Editorfenster vorhandene Programmtext kompiliert.

Befindet sich das zu kompilierende Programm in Fenster 2, führt danach ein Rücksprung zum Editor in Fenster 1; mit <CTRL><SHIFT><2> dann zu Fenster 2 schalten.

Findet der Compiler beim Kompilieren einen Fehler, werden Fehlernummer und die Textzeile, in der sich der Fehler befindet, im Infofeld ausgegeben. Danach übergibt der Compiler die Kontrolle an den Monitor.

---

19) Kapitel III, Abschnitt 2.5 – Optionsmenü.

Beispiele

C<RETURN>
Das im Editor befindliche Programm wird kompiliert.

C "PROGNAME.ACT"
Das Programm PROGNAME.ACT wird vom aktiven Laufwerk aus dem aktuellen Verzeichnis geladen und kompiliert. Bei Derivaten von ATARI DOS 2 ist das Laufwerk #1.

C "D8:PROGNAME.ACT"
Das Programm wird von DOS-Laufwerk #8 kompiliert.

C "DL:>ACTION>PROGNAME.ACT"
Das Programm aus dem Verzeichnis "ACTION" auf Laufwerk "DL:“ (#12) wird kompiliert.

Wichtige Hinweise: Tritt beim Kompilieren ein Fehler #3 auf, hängt sich der Editor beim anschließenden Aufruf auf. Deshalb zuvor in der Monitorzeile 'SET $E=$491^' eingeben, um den Memory Pointer zurückzusetzen; danach erst in den Editor gehen.

Wurde ein Programm bereits erfolgreich kompiliert und springt man vor dessen Ausführung trotzdem zurück in den Editor, wird der kompilierte Code gelöscht. Das Programm muss dann erneut kompiliert werden, bevor es gestartet werden kann.

2.3 Der Sprung ins DOS

Mit D<RETURN> gelangt man ins DOS.

DOS und Programme nutzten oft die gleichen Speicherbereiche wie Action!. Deshalb vor dem Sprung ins DOS zuerst Programmtext und kompilierte Programme abspeichern. Eine 'Memory Save'-Funktion ist dabei vorteilhaft, da diese auch die Parameter des Optionsmenüs speichert[20].

2.4 Aufruf des Editors

Der Editor wird mit E<RETURN> aufgerufen.

Trat beim Kompilieren eines Programms ein Fehler auf, steht der Cursor beim nächsten Aufruf des Editors an der Stelle im Programmtext, an welcher der Fehler entdeckt wurde, oft aber auch in der Zeile nach der Fehlerstelle. Also sorgfältig prüfen.

2.5 Das Optionsmenü

Per Optionsmenü lassen sich die Arbeitsparameter von Monitor, Compiler und Editor verändern. In dieses Menü gelangt man mit O<RETURN>. Jeder einzelne Parameter wird nacheinander in der Kommandozeile angezeigt. Soll eine Option geändert werden, gibt man den neuen Parameter ein und drückt <RETURN>. Soll ein Parameter unverändert bleiben, reicht <RETURN>. Verlassen ist jederzeit durch <ESC> möglich.

Die Beschreibung der Optionen[21] enthält die in der Kommandozeile ausgegebene Information, die voreingestellten Parameter und die Komponenten des Systems, die von der betreffenden Option beeinflusst werden (Monitor=M, Compiler=C, Editor=E).

20) SpartaDOS X - CAR Command.
21) Eine Übersichtstabelle der Optionen befindet sich im Anhang F.

Display? Y (M,C,E)
Der Bildschirm kann während des Ladens und Speicherns sowie beim Kompilieren abgeschaltet werden, um die Arbeitsgeschwindigkeit zu erhöhen. Abschalten mit 'N'.

Bell? Y (M,C,E)
Der Summer ertönt immer dann, wenn ein Fehler im Editor, Compiler oder Monitor auftritt. Außerdem ertönt er beim Aufruf des Monitors. Abschalten mit 'N'.

Case sensitive? N (C)
Wird die Option mit 'Y' (Ja) aktiviert, unterscheidet der Compiler zwischen Groß- und Kleinbuchstaben. Die Variable ZAEHLER ist dann nicht mehr identisch mit der Variablen Zaehler. Außerdem müssen dann alle Schlüsselwörter[22] zwingend in Großbuchstaben geschrieben werden. Zur Vereinfachung hat diese Option die Grundeinstellung 'N'.

Trace? N (C)
Ist die Option eingeschaltet ('Y'), wird auf dem Bildschirm jede aufgerufene Routine zusammen mit den dort verwendeten Parametern ausgegeben[23]. So kann das Kompilieren schrittweise verfolgt werden.

List? N (C)
Bei Eingabe von 'Y' wird die jeweils gerade vom Compiler bearbeitete Programmzeile im Infofeld ausgegeben.

Window 1 size: 18 (E)
Festlegen der Größe von Fenster #1. Da der Bildschirm maximal 23 Zeilen darstellen kann, ist die Größe des Fensters #2 von der des Fensters #1 abhängig. Die maximale Größe jedes Fensters beträgt 18, die minimale Größe 5 Zeilen. Man gibt einfach die gewünschte Zeilenzahl für Fenster #1 ein und drückt <RETURN>. Eine Eingabe, die größer als 18 ist, wird auf 18 gesetzt. Eine Eingabe, die kleiner als 5 ist, wird auf 5 gesetzt.

Anmerkung: Ist nur Fenster #1 aktiv, nimmt es den ganzen Bildschirm ein. Erst wenn Fenster #2 aktiviert wird, wird die eingestellte Größe von Fenster #1 sichtbar.

Line size: 120 (E)
Mit der Zeilenlänge wird die Anzahl der Zeichen festgelegt, die maximal in eine Zeile geschrieben werden können. Gedacht ist diese Funktion vor allem zur Abstimmung auf den für das Listing verwendeten Drucker. Ist das Zeilenende erreicht, ertönt ein Warnton. Die gewünschte Zeilenlänge wird als Zahl eingegeben.

Der Editor kann höchstens 240 Zeichen[24] in einer Zeile darstellen. Wird eine Zahl größer als 240 eingegeben, erfolgt weder eine Korrektur noch erscheint eine Fehlermeldung! Der Editor schneidet einfach jede Zeile nach Zeichen 240 ab. Der Rest ist dann verloren.

Anmerkung: Eine zu große Zeilenlänge im Editor kann ggf. das System zum Absturz bringen. Die voreingestellte Zeilenlänge von 120 Zeichen sollte nicht überschritten werden. Speichern, laden und kompilieren werden dann ggf. kompromittiert.

---

22) Kapitel IV, Abschnitt 2.
23) Mehr dazu im Kapitel IV, Abschnitt 6.4.
24) Tipp dazu in Anhang J

Left margin: 2 (M,E)
Die Voreinstellung wird aus Speicherstelle 82 ($52) übernommen. Zum Ändern gibt man die gewünschte Zahl ein und drückt <RETURN>. Der Wert bleibt auch nach dem Verlassen von Action! erhalten[25]. Ein Wert größer als 22 führt zur Fehlbedienung des Optionsmenüs und der Monitor kann dabei abstürzen. Dann hilft nur noch <RESET>!

EOL character: (Leerzeichen) (E)
Das Zeichen, welches das Zeilenende markiert (End Of Line), wird vom Editor am Zeilenende angezeigt. Voreingestellt ist das vom ATARI-Editor her gewohnte Leerzeichen. Aber man kann hier jedes sichtbare Zeichen eingeben; danach <RETURN> drücken. Ein sichtbares EOL-Zeichen erleichtert das Löschen desselben.

2.6 PROCEED - gestopptes Programm weiterlaufen lassen

Wurde das Programm mit <BREAK> bei Ausführung der BREAK-Routine aus der Runtime Library unterbrochen, kann es mit P<RETURN> so fortgesetzt werden, als ob keine Unterbrechung stattgefunden hätte.

2.7 RUN - Programm laufen lassen

Mit dem Kommando 'Run' wird ein kompiliertes Programm aus dem Monitor heraus gestartet. Folgende Möglichkeiten bieten sich dabei:

R<RETURN> startet ein kompiliertes Programm, das sich im Speicher befindet.

R "<filespec>"<RETURN> lädt den Programmtext vom Speichergerät, kompiliert ihn und startet danach das fertig kompilierte Programm[26]. Beispiele:

R "D8:PROGNAME.ACT"<RETURN> - Text von Laufwerk #8 laden, kompilieren und starten.

R "PROGNAME.ACT"<RETURN> - Text vom aktiven Laufwerk aus dem aktuellen Verzeichnis laden, kompilieren und Programm starten.

R <Adresse><RETURN> startet bereits kompilierte Programme oder Routinen an der angegebenen Adresse. Das ist hilfreich bei der Fehlersuche in einem Programm, das eine Routine in Maschinensprache aufruft. Beispiele:

R $0600<RETURN> - Programm an Adresse $0600 starten.
R 1024<RETURN> - Programm an Adresse 1024 starten.

R <Routine><RETURN>[27] startet eine Routine aus einem kompilierten Programm. Dabei muss es sich um einen gültigen PROC- oder FUNC-Namen handeln.

Nach der Ausführung wird die Kontrolle an den Monitor zurückgegeben. Bei besonders schweren Fehlern (z.B. nicht geschlossene Schleife) stürzt Action! allerdings ab. Dann hilft nur noch <RESET>[28]. So führt z.B. die Eingabe von 'R *<RETURN> zum Absturz.

---

25) Ausnahme: SDX ab V. 4.47 stellt LMARGN auf den mit SCRDEF gesetzten Wert zurück.
26) Die genauen Vorgaben zur <filespec> bitte im jeweiligen DOS-Handbuch nachschlagen.
27) Dabei können keine Parameter übergeben werden!
28) Weitere Informationen zum Programmablauf enthält Kapitel III, Abschnitt 3.

2.8 SET - Setzen eines Werts in eine Speicherstelle

Das Kommando SET funktioniert im Monitor wie unter Sprache beschrieben. Die Erläuterung dazu befindet sich im Kapitel IV, Abschnitt 7.3.

2.9 WRITE - Abspeichern eines kompilierten Programms

Ein kompiliertes Programm (Binärdatei) wird mit 'WRITE' abgespeichert. Es kann später von DOS aus geladen und gestartet werden[29]. Ist auf einem Medium nicht ausreichend Speicherplatz vorhanden oder ist es schreibgeschützt, wird eine entsprechende Fehlermeldung ausgegeben.

Beispiele für das WRITE-Kommando

W "<filespec"<RETURN> - speichern eines kompilierten Programms auf Medium. Die Vorgaben des verwendeten DOS beachten. Beispiele:

W "D8:PROGNAME.ACT"<RETURN> - kompiliertes Programm auf Laufwerk #8 im aktuellen Verzeichnis speichern.

Anmerkungen: Das DOS-Kommando zum Laden und Starten eines Programms in Maschinensprache kann für die mit 'WRITE' abgespeicherten Binärdateien benutzt werden. Dabei muss das Modul eingesteckt sein, da zumeist Routinen aus der Library benötigt werden. Soll das Programm ohne Modul verwendet werden, muss eine 'Runtime Library' beim Kompilieren in das Programm eingebunden werden.

Sollte der WRITE-Befehl aufgrund eines Disk-Fehlers nicht erfolgreich ausgeführt werden, wird der IOCB nicht korrekt geschlossen. Wechselt man dann vor Ausführung der nächsten Disk-Operation die Disk, werden ungültige Daten auf die neue Disk geschrieben. Tritt ein derartiger Fehler auf, hilft die Eingabe von 'X Close(1) <RETURN>' im Monitor[30]. Danach kann die fehlerhafte Datei von DOS aus gelöscht werden.

2.10 XECUTE - Sofort ausführbare Kommandos

Jede Action!-Programmanweisung und die Compiler-Direktiven mit Ausnahme von MODULE und SET können direkt aus dem Monitor ausgeführt werden. Dazu braucht man jeweils nur die Anweisung 'XECUTE' voranzustellen.

Beispiele X PrintE("Hallo Leute")<RETURN>[31]
X trace[32] = 255<RETURN>

2.11 ? - Inhalt einer Speicherstelle anzeigen

Mit dieser Funktion kann man sich entweder den Inhalt einer Speicherstelle oder den Wert einer Variablen auf dem Bildschirm ausgeben lassen. Das Kommando dazu lautet

?<Compilerkonstante><RETURN>.

Die Eingabe ist sowohl in dezimaler als auch hexadezimaler Schreibweise zulässig. Bei Verwendung von Hexadezimalzahlen sind ausschließlich Großbuchstaben gültig. Kleinbuchstaben verursachen einen Systemfehler und/oder werden ignoriert.

---

29) **Wichtig:** Siehe Hinweis in Kapitel I, Abschnitt 4.
30) Library Routine Close: Siehe Kapitel VI, Abschnitt 4.2.
31) Hier können Parameter übergeben werden; z.B.: X TEST(4).
32) So aktiviert wird die TRACE-Option erst nach erneutem Kompilieren des Programms wirksam.

Mit ?$700<RETURN> wird beispielsweise der Inhalt der Speicherstelle $700 angezeigt. Die Ausgabe lautet:

```
1792,$0700 = s $4453 83 17491
```

Der Monitor gibt die angegebene Adresse in dezimaler und hexadezimaler Schreibweise wieder. Nach dem '=' stehen

- der Wert der Speicherstelle, dargestellt als ATASCII-Zeichen,
- der Hexadezimalwert der CARD, die ab dieser Speicherstelle berechnet wird (2-Byte-Wert),
- der Dezimalwert des BYTEs, das hier gespeichert ist,
- und der Dezimalwert der CARD, die ab dieser Speicherstelle berechnet wird (2-Byte-Wert).

Eine Variable wird abgefragt mit:

?<Kennung><RETURN>.

Ist die Kennung der Variablen nicht in der Symboltabelle[33] des Compilers eingetragen, erscheint die Fehlermeldung:

"Error: 8" (Variable ist nicht definiert)

Die ausgegebenen Werte entsprechen nicht immer unbedingt den Erwartungen, da sie durch das Kompilieren verändert werden.

2.12 * - Speicherinhalt ausgeben

Beginnend ab der angegebenen Speicherstelle wird fortlaufend der Inhalt jeder Speicherstelle auf dem Bildschirm ausgegeben. Die Form der Anzeige ist identisch mit der in Abschnitt 2.11 beschriebenen. Die Anweisung dazu lautet:

*<Adresse><RETURN>

Der Monitor gibt eine fortlaufende Liste in der angesprochenen Form aus. Jede Zeile bezieht sich nur auf eine Speicherstelle. Stoppen lässt sich diese Funktion mit der Leertaste. Anhalten lässt sich das Listen mit <CTRL><1>, nochmaliges Drücken lässt das Listen weiterlaufen.

3 Funktionen zur Fehlerbeseitigung

Ein Programm läuft manchmal nicht so wie erwartet. Das muss nicht unbedingt an einem Fehler liegen. Viel wahrscheinlicher ist, dass die Konzeption in nicht geeignete Routinen umgesetzt wurde. Doch der Monitor mit seinem Optionsmenü erlaubt eine schrittweise Ausführung des Programms und damit eine Lokalisierung des Problems oder Fehlers.

---

33) Kapitel V, Abschnitt 2.6.

Die Option TRACE
Ist diese Option mit 'Y' eingeschaltet, lässt sich die Programmausführung verfolgen. Dann werden nämlich der Name jeder aufgerufenen Routine und die von ihr beeinflussten Parameter auf dem Bildschirm ausgegeben. Auf diese Weise kann man meist sehr schnell feststellen, wo das Problem liegt. Wenn das klappt, prima! Wenn nicht, muss man zu anderen Tricks greifen.

Um weitere Tricks anwenden zu können, muss man erst mal die Programmausführung unterbrechen. Im Monitor gibt es dafür zwei Möglichkeiten. Die <BREAK>-Taste und die Library-Routine 'Break'.

Die <BREAK>-Taste
Anders als im Editor steht die <BREAK>-Taste im Monitor zur Verfügung. Allerdings gibt es im Gebrauch einige Einschränkungen. Mit <BREAK> kann ein Programm nur unterbrochen werden

- während einer Ein-/Ausgabe-Operation,
- bei Aufruf einer Routine mit mehr als 3 Parametern.

Das scheinen strenge Rahmenbedingungen zu sein, für die es aber sehr gute Gründe gibt. Das Action!-System selbst prüft nicht, ob während einer Programmausführung <BREAK> gedrückt wurde. Doch das System springt in den zwei oben genannten Fällen in die CIO. Und die CIO prüft, ob <BREAK> gedrückt wurde.

Library PROC Break()
Will man das Programm an irgendeiner Stelle unterbrechen, ruft man einfach diese Library-Routine an der betreffenden Stelle auf. Die Routine arbeitet exakt so wie die <BREAK>-Taste, funktioniert aber unter allen Bedingungen. Diese Methode zum Abbrechen eines Programms ist wesentlich zuverlässiger, als <BREAK> zu drücken. Man weiß nämlich vorher genau, an welcher Stelle das laufende Programm unterbrochen wird. Diese Routine kann auch mehrmals im Programm eingesetzt werden, wenn es an verschiedenen Stellen unterbrochen werden soll.

Hat das Programm gestoppt, kann man mit den Monitorkommandos '*' und '?' die Werte der benutzten Variablen überprüfen. Wird diese Methode der Fehlersuche zusammen mit der TRACE-Funktion verwendet, weiß man jederzeit, an welcher Stelle im Programm man sich befindet. So können lokale und globale Variablen in der PROC gleichermaßen geprüft werden. Reicht diese Methode wider erwarten nicht aus, kann man noch 'Print'-Anweisungen zum Prüfen im Programm einfügen, um z.B. eine nicht funktionierende Schleife zu entfehlern. Beispiel:

```
PrintE("In Schleife FOR x=1 to 100")
PrintBE(x)
```

IV. Action! - Die Sprache

1 Einführung
Die Sprache ist das Herz des Action!-Systems. In die Sprache wurden die Vorteile von C und Pascal übernommen, und sie ist die schnellste Hochsprache, die es für ATARI 8-Bit-Rechner gibt. Wer bereits über Kenntnisse in BASIC oder anderen weniger strukturierten Sprachen verfügt, wird Action! als merkliche Verbesserung empfinden. Aufgrund ihrer Struktur arbeitet die Sprache ähnlich wie unser Verstand, wenn er Ideen entwickelt. Der Aufbau eines Programms ergibt sich einfach beim Lesen, ohne dass man sich durch etliche GOTOs und nicht deklarierte Variablen wühlen muss.

Programme in Action! zu strukturieren ist einfach, weil sie aus einzelnen Routinen Stück für Stück zusammengesetzt werden. Diese Routinen bestehen aus einer Anzahl von zusammengehörigen Anweisungen, die eine festgelegte Aufgabe erfüllen sollen. Sind nun die Komponenten für alle Anforderungen geschrieben, die das Programm erfüllen soll, ist es ziemlich einfach, diese vom Rechner ausführen zu lassen. Es entspricht in etwa einer Liste von Arbeitsschritten wie z.B.:

1. Betten machen.
2. Zimmer aufräumen.
3. Wohnzimmer lüften.
4. Einkaufen gehen.

Der Rechner führt die Arbeitsschritte in der Reihenfolge aus, in der sie aufgelistet sind und nicht in der, die für uns am sinnvollsten wäre.

Einzelne Routinen machen es leicht, eine einzelne Aufgabe wieder und wieder abzuarbeiten oder diese Aufgabe mehrfach in verschiedenen Programmteilen unter anderen Bedingungen auszuführen.

Die einzige Voraussetzung für diese strukturierte Art der Problemlösung ist: Das Programm muss aus gültigen Routinen bestehen, damit es funktioniert. Diese werden in Action! Prozeduren (PROC) und Funktionen (FUNC) genannt. Ein Programm besteht meist aus mehreren Routinen.

Wichtig: Wird ein Programm aus mehreren Routinen kompiliert und gestartet, wird die letzte Routine von Action! als Hauptroutine angesprochen. Daher sollte diese zum Steuern des Programms verwendet werden.

2 Action! - Der Sprachumfang
Für die Erläuterungen im Kapitel IV sind einige begriffliche Festlegungen erforderlich, damit ähnliche aber inhaltlich verschiedene Begriffe später vom Anwender sicher auseinandergehalten werden können. Weitere im Abschnitt 2.1 nicht erläuterte Fachbegriffe werden im Kapitel IV dort erläutert, wo sie zum ersten Mal Anwendung finden.

2.1 Spezielle Begriffe
Zur Erläuterung der Sprache werden einige Begriffe benutzt, mit denen möglicherweise nicht jeder vertraut ist. Nachfolgend Erläuterungen zu den verwendeten Begriffen.

Adresse
Eine Adresse ist eine Speicherstelle. Soll der Rechner etwas in den Speicher packen, muss ihm dazu, ähnlich wie auf einem Brief, eine Adresse angegeben werden. Allerdings

benötigt der Rechner nur eine "Hausnummer", daher ist eine Adresse für den Rechner nur eine Zahl.

Alphabetisch
Umfasst alle Buchstaben des Alphabets, also Groß- (ABC) und Kleinbuchstaben (abc). "Alphanumerisch" beinhaltet zusätzlich die Ziffern "0" bis "9" (z.B. a9B).

Kennung
In diesem Handbuch werden Namen, die Variablen, PROCs etc. gegeben werden, als Kennung bezeichnet. Dies ist notwendig, weil Namen in Action! bestimmten Regeln unterliegen:

- Sie müssen mit einem Buchstaben beginnen.
- Die restlichen Zeichen müssen alphanumerisch sein; erlaubt ist auch der Unterstrich (_).
- Sie dürfen keine Schlüsselwörter sein.

Diese Regeln müssen unbedingt eingehalten werden, wenn eine Kennung festgelegt wird. Andernfalls führt das zu einem Syntax-Fehler.

MSB, LSB
MSB steht für "Most Significant Byte" (höherwertiges Byte), LSB für "Least Significant Byte" (niederwertiges Byte). Das Dezimalsystem besteht aus nach Stellenwert geordneten Ziffern, nicht aus Bytes. So ist zum Beispiel die höherwertige Ziffer der Zahl '54' die '5', die niederwertige '4'. Man merke sich also, dass in Action! Zwei-Byte-Zahlen im für 6502-Rechner üblichen LSB-MSB-Format gespeichert werden.

$
Wird einer Zahl ein Dollarzeichen vorangestellt, teilt das dem Rechner mit, dass es sich um eine Hexadezimalzahl handelt. Hexadezimalzahlen sollten immer dann angewendet werden, wenn der Computer direkt programmiert wird.

Beispiele:

```
24FC
$0D
$88
$F000
```

;
Das Semikolon kündigt einen Kommentar an. Alles, was in einer Zeile nach dem Semikolon steht, wird vom Compiler ignoriert.

Beispiele:

```
; Dies ist ein Kommentar.
Dies ist keiner und ruft einen Fehler beim Kompilieren
hervor
; Dieser Kommentar beinhaltet
; sogar ein ; Semikolon
var=3      ; Kommentare dürfen auch hinter
; auszuführenden Anweisungen stehen.
; Ein dreizeiliger Kommentar
;
; mit einer Leerzeile.
```

< und >
Was immer auch zwischen diesen beiden spitzen Klammern steht, ist stets ein Platzhalter, aber niemals ein Schlüsselwort. Üblicherweise wird durch einen Platzhalter angegeben, was in einem Programm dort als Angabe notwendig wäre. Wird z.B. eine <Kennung> verlangt, bedeutet das, dass eine gültige Kennung benutzt werden muss.

{ und }
Was zwischen diesen beiden geschweiften Klammern steht, ist als Option möglich, aber nicht zwingend erforderlich. {Kennung} bedeutet z.B., dass eine gültige Kennung an dieser Stelle benutzt werden kann, diese aber für den Compiler nicht erforderlich ist.

l: und :l
Wie in der Musik bedeuten diese Zeichen eine Wiederholung. Was dazwischen steht, kann ab null Mal wiederholt werden. l:Kennung:l bedeutet z.B., dass an dieser Stelle eine Liste von null oder mehr Kennungen erlaubt ist.

l
Dieses Symbol zeigt eine 'Oder'-Situation an. <Kennung>l<Adresse> bedeutet, dass entweder eine Kennung oder eine Adresse verwendet werden kann, aber nicht beides.

2.2 Schlüsselwörter

In Action! werden wie in anderen Programmiersprachen auch Schlüsselwörter verwendet. Ein Schlüsselwort ist für Action! jedes Wort oder Symbol, das der Compiler als etwas Besonderes erkennt. Entweder handelt es sich dann um einen Operator, den Namen eines Datentyps, eine Anweisung oder eine Compiler-Direktive.

Zusammen mit den Erklärungen zu den einzelnen Schlüsselwörtern wird auch die dazu gehörende Syntax[34] dargestellt und erläutert.

Und hier nun die Auflistung der Schlüsselwörter:

AND	FI	OR	UNTIL	=	(
ARRAY	FOR	POINTER	WHILE	<>	)
BYTE	FUNC	PROC	XOR	#	.
CARD	IF	RETURN	+	>	[
CHAR	INCLUDE	RSH	-	>=	]
DEFINE	INT	SET	*	<	"
DO	LSH	STEP	/	<=	'
ELSE	MOD	THEN	&	$	;
ELSEIF	MODULE	TO	%	^	
EXIT	OD	TYPE	!	@	

Warnung: Die Schlüsselwörter ausschließlich in der beschriebenen Weise verwenden!

34) Anlage A – Sprachsyntax von Action!, dargestellt in Backus-Naur-Form.

3 Die elementaren Datentypen

Zuerst ist einiges zu Variablen und Konstanten zu sagen, da sie die eigentlichen vom Rechner bearbeiteten Datenobjekte sind. Danach werden die Datentypen erläutert.

3.1 Variablen

Es sind ausschließlich gültige Kennungen als Variablennamen erlaubt. Dies ist aber auch die einzige Beschränkung bei der Wahl eines Variablennamens. Das momentan noch fehlende Wissen über die Arbeitsweise von Funktionen (FUNC) und Variablen (VAR), um die Möglichkeiten einer Variablen darlegen zu können, wird in Abschnitt 6.3 behandelt.

3.2 Konstanten

In Action! sind drei Arten von Konstanten möglich: numerische, String (Text)- und Compiler-Konstanten.

Numerische Konstanten können in drei verschiedenen Formaten eingegeben werden:

- Hexadezimal,
- Dezimal oder
- als Zeichen.

Hexadezimale Konstanten werden durch ein vorangestelltes Dollarzeichen ($) markiert.

Beispiele:

```
$4A00
$0D
$300
```

Dezimale Konstanten brauchen keine spezielle Markierung.

Beispiele:

```
65500
2
324
```

Anmerkung: Sowohl hexadezimale als auch dezimale Konstanten können ein negatives Vorzeichen haben, also:

```
-$8c
-4360
```

Zeichenkonstanten werden durch ein vorangestelltes Hochkomma (') gekennzeichnet. Zeichen sind deshalb numerische Konstanten, weil sie rechnerintern als Ein-Byte-Zahlen verarbeitet werden. Der gesamte ATASCII-Zeichensatz steht dafür zur Verfügung.

Beispiele:

```
'A
'@
'"
```

Stringkonstanten bestehen aus einem String (Zeichenkette) von null (!) oder mehr Zeichen, die in Anführungszeichen (") eingeschlossen sind. Im Speicher wird jedem String die dazugehörige Länge vorangestellt. Die Anführungszeichen werden nicht als Teil des Strings akzeptiert. Sollen Anführungszeichen in einem String vorhanden sein, muss man zwei nebeneinandersetzen.

Beispiele:
```
"Dies ist eine Stringkonstante"
"so""kriegt man ein Anführungszeichen hinein"
"58395"
"q" (Stringkonstante mit einem Zeichen!)
```

Compiler-Konstanten unterscheiden sich von den bisherigen Konstantenarten darin, dass sie nur beim Kompilieren eines Programms benötigt werden, um Attribute zu VARs, PROCs, FUNCs und Code-Blöcken zu setzen. Im fertigen Programm werden sie nicht berücksichtigt. Folgende Formate sind hier erlaubt:

- Eine numerische Konstante.
- Eine vordefinierte Kennung.
- Verweis auf einen Zeiger.[35]
- Die Kombination von zweien der drei ersten Formate.

Bisher wurde nur das erste Format angesprochen. Nun folgen die drei anderen Formate.

Wird in einer Compiler-Konstante eine bereits vordefinierte Kennung verwendet (z.B. ein VAR-, PROC- oder FUNC-Name), enthält der benutzte Wert die Speicheradresse der Kennung.

Das dritte Format ermöglicht das Setzen von Zeigern per Compiler-Konstante.

Das letzte Format erlaubt eine Kombination von zwei der drei ersten Formate per einfacher Addition. Nachfolgende einige Beispiele für zulässige Formate:

```
cat    ; benutzt die Adresse der Variablen 'cat'
$8D00  ; eine Hexadezimalkonstante
dog^   ; ein 'Zeiger' auf einen Zeiger als Konstante
5+ptr^ ; 5 plus Inhalt des Zeigers 'ptr'
$80+p  ; geht auf $80 plus Adresse von 'p'
```

3.3 Elementare Datentypen

Mithilfe von Datentypen gibt der Programmierer dem Strom von Bits und Bytes, die der Computer verarbeitet, erst einen Zweck. So wird es erst möglich Konzepte anzuwenden, die man versteht. Daher braucht man nicht zu wissen, wie der Rechner wirklich arbeitet. Action! unterstützt drei elementare Datentypen sowie flexible Erweiterungen[36] dazu.

Es gibt die Typen BYTE, CARD und INT, von denen jeder Einzelne nachfolgend ausführlich erläutert wird. Alle sind numerisch, sodass bei der Eingabe von Daten das numerische Format verwendet werden muss.

3.3.1 BYTE

BYTE wird für positive Integerzahlen (ganze Zahlen) benutzt, die kleiner als 256 sind. Intern wird es als Ein-Byte-Zahl ohne Vorzeichen (unsigned) dargestellt. Der Wertebereich reicht von 0 bis 255. Auf den ersten Blick scheint das eine nutzlose Datentype zu sein, aber sie hat zwei wertvolle Eigenschaften.

---

35) Kapitel IV, Abschnitt 8.1.
36) Kapitel IV, Abschnitt 8.

Als Zähler in Schleifen (WHILE, UNTIL, FOR) beschleunigt BYTE den Programmablauf merklich, weil es für den Rechner einfacher ist, ein Byte statt vieler zu bearbeiten. Und da Zeichen intern als Ein-Byte-Zahlen dargestellt werden, ist BYTE auch als Zeichentyp gut verwendbar. Das rührt daher, dass der Compiler die Schlüsselworte BYTE und CHAR als gleichwertig, also austauschbar behandelt. Wer Erfahrung im Umgang mit Pascal oder C besitzt, dem wird diese Art des Umgangs mit Zeichen vertraut sein.

3.3.2 CARDinal

CARD ist dem Typ BYTE sehr ähnlich, nur mit dem Unterschied, dass er wesentlich größere Zahlen beinhalten kann. Da CARD intern als Zwei-Byte-Zahl ohne Vorzeichen (unsigned) dargestellt wird, lassen sich damit Werte von 0 bis 65.535 verarbeiten. Eine CARD wird im für 6502-Maschinen gebräuchlichen LSB-MSB-Format gespeichert.

3.3.3 INTeger

Dieser Datentyp kann nur Integerzahlen beinhalten und wird ebenfalls im numerischen Format eingegeben. INT kann sowohl negative als auch positive Integerzahlen im Wert von -32768 bis 32767 beinhalten und wird intern als Zwei-Byte-Zahl mit Vorzeichen (signed) im LSB-MSB-Format gespeichert.

3.4 Deklarationen

Mit Deklarationen wird dem Rechner mitgeteilt, dass etwas festgelegt wird. Soll z.B. die Variable 'Kosten' als Typ CARD definiert werden, muss das der Rechner irgendwie erfahren. Andernfalls kann der Computer mit der Information 'Kosten' nichts anfangen.

Jede gewünschte Kennung muss vor dem erstmaligen Gebrauch deklariert werden, egal ob es ein VAR-, PROC-, oder FUNC-Name ist. Die Deklaration von VARs wird hiernach erklärt, gefolgt von einer Anmerkung zur Deklaration von numerischen Konstanten. Die Deklaration von PROCs und FUNCs wird im Abschnitt 6 erläutert.

3.4.1 Deklaration von Variablen

Der Vorgang bei der Deklaration von Variablen ist bei den elementaren Datentypen immer gleich. Das grundsätzliche Format dafür ist:

<Typ> <Kennung>{=<Initinfo>} l:,<Kennung>{=<Initinfo>}:l[37]

wobei

<Typ>	der elementare Datentyp der zu deklarierenden Variable(n) ist.
<Kennung>	die Kennung zur Benennung der Variablen ist.
<Initinfo>	es ermöglicht, den Wert einer Variablen zu initialisieren oder die Speicherstelle für die Variable festzulegen.
'<Initinfo>'	hat dabei die Form <Adr> l [<Wert>].

Dabei ist

<Adr>	die Adresse der Variablen und muss eine Compiler-Konstante sein.
<Wert>	der Startwert der Variablen[38] und muss eine numerische Konstante sein.

37) Erläuterung zu <, >, {, }, l:, :l, und l im Kapitel IV, Abschnitt 2 und Anhang A
38) Der Startwert wird beim Kompilieren gesetzt und gilt nur für den ersten Programmdurchlauf.

Mit <Typ> lässt sich mehr als eine Variable deklarieren. Zusätzlich kann dem Compiler mitgeteilt werden, wo eine Variable im Speicher stehen soll oder welchen Anfangswert sie annehmen soll. Beispiele zu den Formaten:

```
BYTE top,hat ; deklariert 'top' und 'hat' als BYTE-Variable

BYTE x=$8000, ; deklariert 'x' als BYTE und platziert es
              ; in Speicherstelle $8000.
     y= [0]   ; deklariere und initialisiere 'y'

CARD ctr= [$83D4], ; deklariert und initialisiert
     bignum= [0],  ; drei Variablen als
     cat= [30000]  ; Typ CARD

INT NUM= [0] ; deklariert 'num' als INT-Variable und
             ; initialisiert den Anfangswert auf 0.
```

Die letzten zwei Beispiele zeigen, dass die Variablen nicht unbedingt in der gleichen Zeile stehen müssen. Der Compiler liest Variablen des vorgegebenen Typs solange ein, wie Kommas als Trennzeichen zwischen ihnen stehen. Also nach der letzten Variablen in der Liste auf keinen Fall ein Komma setzen. Das kann ungeahnte Folgen haben.

Die Deklaration von Variablen muss unmittelbar nach einer MODULE-Anweisung[39] oder am Beginn einer PROC oder FUNC[40] erfolgen. An anderen Stellen im Programm führt das zu einem Fehler[41].

Anmerkungen: Hat man für eine globale oder lokale Variable eine Adresse festgelegt, kann man ihr einen Wert nur noch in einer FUNC oder PROC übergeben.

Wurde einer Variablen einen Startwert z.B. mit "BYTE VAR=[xx]" zugewiesen (Initialisierung), dann kann für diese Variable keine Adresse mehr festgelegt werden.

Die Initialisierung per Deklaration bewirkt, dass dieser Wert nur beim Kompilieren einmal gesetzt wird und im Programm nur solange gilt, wie er nicht verändert wird.

Wird der Wert einer Variablen (BYTE, CARD oder INT) im Programmablauf verändert und soll sie dennoch beim Aufruf der entsprechenden PROC oder FUNC immer einen bestimmten Anfangswert haben, darf sie nicht bereits bei ihrer Deklaration initialisiert werden, sondern erst an geeigneter Stelle in einer PROC oder FUNC.

Beispiel1

```
MODULE

BYTE a=709, b=710  ; Farbregister für Schrift u. Hintergrund
   :
PROC farben()
```

39) Kapitel IV, Abschnitt 7.4.
40) Kapitel IV, Abschnitt 6.1.1 und 6.2.1.
41) Siehe auch Kapitel IV, Abschnitt 6.3, Geltungsbereiche.

```
  a=10
  b=4
    :
    :
RETURN
    :
```

Die globale Variablen a und b vom Typ BYTE werden auf die Speicherstellen 709 und 710 festgelegt. Werte werden ihnen erst in der PROC 'farben' zugewiesen, wodurch die Farben dann entsprechend gesetzt werden. Sie gelten für alle Routinen im Programm und können daher in jeder Routine einen Wert zugewiesen bekommen.

Beispiel 2

```
  :
PROC farben()

  BYTE a=709, b=710  ; Farbregister für Schrift u. Hintergrund
  a=10
  b=4
    :
    :
RETURN
    :
    :
```

Hier werden die Variablen a und b als lokale Variablen auf die Speicherstellen 709 und 710 festgelegt. Sie gelten nur innerhalb der PROC 'farben' und können anders als in Beispiel 1 nicht in den anderen Routinen des Programms verwendet werden.

3.4.2 Numerische Konstanten

Numerische Konstanten werden nicht extra deklariert. Ihr Typ wird durch die Anwendung deklariert. Ist eine numerische Konstante kleiner als 256, wird sie automatisch als Typ BYTE deklariert, andernfalls als CARD. Aus praktischen Gründen wird eine negative Konstante (z.B. -7) als Typ INT behandelt.

Konstante	Typ
543	CARD
$0D	BYTE
$F42	CARD
'W	BYTE

4 Ausdrücke

Ausdrücke sind Konstruktionen, durch die man die Werte von Variablen, Konstanten und Bedingungen durch Benutzen bestimmter Operatoren erhält. Zum Beispiel entspricht der Ausdruck '4+3' dem Wert '7', solange der Operator '+' in diesem Ausdruck für Addition benutzt wird. Ist der Operator stattdessen '*', ergibt das eine Multiplikation und der Ausdruck entspricht dem Wert '12' (4*3=12).

Action! Verfügt über zwei Typen von Ausdrücken: arithmetische und relationale. Das obige Beispiel ist ein arithmetischer Ausdruck. Relationale Ausdrücke bringen als Ergebnis eine 'richtige' oder 'falsche' Antwort. '5>=7' ist falsch, wenn '>=' benutzt wird, um 'ist größer als oder gleich' auszudrücken. Dieser Typ wird angewendet, um bedingte Anweisungen zu bearbeiten[42]. Eine alltägliche bedingte Anweisung wäre z.B.: „Ist es bereits 5 Uhr oder später, dann ist es Zeit, nach Hause zu gehen." Ein relationaler Ausdruck in Action! kann so lauten:

Stunde >= 5

Diese Überprüfung erfolgt - wie viele andere auch - automatisch beim Blick auf die Uhr. Der Computer allerdings braucht exakte Angaben zu dem, was er überprüfen soll.

Bevor Ausdrücke weiter vertieft werden können, müssen zuerst die Operatoren dargelegt werden, die zu jedem Typ gehören. Dann folgt eine Diskussion jedes Ausdrucks und weiter geht es mit den speziellen Erweiterungen der relationalen Ausdrücke.

4.1 Operatoren

Action! unterstützt drei Arten von Operatoren:

- Arithmetische Operatoren
- Bitweise Operatoren
- Relationale Operatoren

Wie schon die Namen der ersten und letzten Operatoren vermuten lassen, gehören sie zu einem speziellen Ausdruckstyp. Die zweite Art von Operatoren führt arithmetische Operationen und Adressoperationen auf Bitebene aus.

4.1.1 Arithmetische Operatoren

Arithmetische Operatoren sind allgemein bereits aus der Mathematik bekannt. Doch sind einige so modifiziert worden, dass man sie von einer Rechnertastatur aus eingeben kann. Hier eine Liste derjenigen Operatoren, die Action! unterstützt, jeweils ergänzt um ihre Bedeutung:

-	minus (negatives Vorzeichen) z.B.: -5
*	Multiplikation, z.B.: 4*3
/	Division von Integerzahlen, z.B.: 13/5. Das Ergebnis ist 2, da der Nachkommaanteil nicht berücksichtigt wird.
MOD	Rest einer Division von Integerzahlen, z.B.: 13 MOD 5. Das ergibt 3, da 13/5=2 Rest 3 ist.
+	Addition, z.B.: 4+3
-	Subtraktion, z.B.: 4-3

Nicht vergessen, dass '=' kein arithmetischer Operator ist. Es wird lediglich in relationalen Ausdrücken, einigen Deklarationen und Zuordnungsanweisungen benutzt.

42) Kapitel IV, Abschnitt 5.2.1.

4.1.2 Bitweise Operatoren

Bitweise Operatoren verarbeiten Zahlen in ihrer binären Form. Das heißt, dass man Operationen ähnlich wie der Computer selbst vornehmen kann (der ja immer mit binären Zahlen arbeitet). Die nachfolgende Liste zeigt die Operatoren:

&	bitweises 'UND'
%	bitweises 'ODER'
!	bitweises 'Exklusiv ODER'
XOR	dasselbe wie "!"
LSH	nach links verschieben
RSH	nach rechts verschieben
@	Adresse von

Die ersten drei Operatoren vergleichen Zahlen Bit für Bit und liefern ein vom Operator abhängendes Ergebnis zurück, so wie nachfolgend dargestellt.

Bitweises UND

<table>
<tr><td rowspan="5">& vergleicht die beiden Bits und gibt ein Ergebnis nach dieser Tabelle zurück</td><td>Bit A</td><td>Bit B</td><td>Ergebnis</td><td>Beispiel</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td rowspan="4">00000101 (dezimal 5)
00100111 (dezimal 39)
&-------
00000101

(Ergebnis von & ist 5)</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
</table>

Bitweises ODER

<table>
<tr><td rowspan="5">% gibt Ergebnisse nach dieser Tabelle zurück</td><td>Bit A</td><td>Bit B</td><td>Ergebnis</td><td>Beispiel</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td rowspan="4">5 % 39
00000101 (5)
00100111 (39)
%-------
00100111 (39)</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
</table>

Bitweises XOR

<table>
<tr><td rowspan="5">! gibt Ergebnisse nach dieser Tabelle zurück</td><td>Bit A</td><td>Bit B</td><td>Ergebnis</td><td>Beispiel</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td rowspan="4">5 ! 39
00000101 (5)
00100111 (39)
!-------
00100010 (34)</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table>

Bitweises Verschieben
Die Operatoren LSH und RSH verschieben Bits. Beim Bearbeiten von Zwei-Byte-Typen (CARD und INT) wird über beide Bytes verschoben. Im Falle von INT wird das Vorzeichen nicht bewahrt und kann sich dabei verändern. Die allgemeine Form lautet:

<Operand> <Operator> <Anzahl der Schiebeoperationen>

wobei

<Operand>	eine numerische Konstante oder Variable ist,
<Operator>	LSH oder RSH ist,
<Anzahl ...>	eine numerische Konstante oder Variable ist, welche die Anzahl der auszuführenden Bit-Verschiebungen bestimmt.

Einige Beispiele zur Verdeutlichung sowohl für LSH als auch RSH:

```
(5)               00000101   (39)              00100111
(5 LSH 1 = 10)    00001010   (39 LSH 1 = 78)   01001110
(5 RSH 1 = 2)     00000010   (39 RSH 1 = 19)   00010011

Operation         MSB        LSB
-----             01010110   11001010          ($56CA)
LSH 1             10101101   10010100          ($56CA LSH 1 - $AD94)
RSH 1             00101011   01100101          ($56CA RSH 1 - $2B65)
LSH 2             01011011   00101000          ($56CA LSH 2 - $5B28)
RSH 2             00010101   10110010          ($56CA RSH 2 - $15B2)
```

Man merke sich einfach, dass LSH um eins der Multiplikation mit 2 entspricht und RSH um eins der Division durch 2 (bei positiven Zahlen). Tatsächlich ist diese Art der Multiplikation bzw. Division schneller als '*2' oder '/2', da sie einfach maschinennah ist. Der Computer muss sie nicht erst in Binärsprache zu übersetzen.

Adress-Operator '@'
Dieser spezielle bitweise Operator liefert die tatsächliche Adresse einer Variablen. Er kann nicht auf Konstante angewendet werden. '@ctr' zeigt die Speicheradresse der VAR 'ctr' an. Der Operator '@' ist sehr hilfreich beim Umgang mit Zeigern.

4.1.3 Relationale Operatoren
Die in nachfolgender Tabelle aufgelisteten Operatoren sind nur in relationalen Ausdrücken zugelassen. Relationale Ausdrücke wiederum sind nur in IF-, WHILE- und UNTIL-Anweisungen erlaubt. Wie schon im Überblick zu diesem Kapitel erläutert, testen relationale Operatoren vergleichende Bedingungen.

(Tabelle umseitig)

Operator	Operation	Beispiel
=	prüft Gleichheit	4=7 (falsch)
#	prüft Ungleichheit	4#7 (richtig)
<>	wie #	5#5 (falsch)
>	prüft auf größer als	9>2 (richtig)
>=	prüft auf größer als oder gleich	5>=5 (richtig)
<	prüft auf kleiner als	2<9 (richtig)
<=	prüft auf kleiner als oder gleich;	6<=6 (richtig)
AND	logisches 'UND'	siehe Abschnitt 4.4
OR	logisches 'ODER'	siehe Abschnitt 4.4

Da die beiden Operatoren '#' und '<>' in Action! gleichbedeutend sind, kann man den benutzen, der vertrauter ist. 'AND' und 'OR' sind spezielle relationale Operatoren und werden in Abschnitt 4.4 unter 'Komplexe relationale Ausdrücke' erklärt.

Technische Anmerkung: Der Compiler stellt Vergleiche an, indem er die beiden Werte voneinander subtrahiert und die Differenz mit null vergleicht. Diese Methode funktioniert bis auf eine Ausnahme korrekt! Der Vergleich von großen positiven mit großen negativen Integer-Werten kann das zu einem falschen Ergebnis führen, weil Integer-Werte das höchstwertige Bit als Vorzeichen-Bit verwenden.

4.1.4 Rangordnung für Operatoren

Operatoren bedürfen einer bestimmten Rangfolge, einer festgelegten Folge zum Abarbeiten oder es wäre unklar, wie der Ausdruck 4+5*3 verarbeitet werden soll.

Entspricht das nun '(4+5)*3' oder '4+(5*3)'? Ohne eine festgelegte Rangfolge ließe sich das nicht sagen. Die eindeutige Grundordnung in Action! ist durch Klammern veränderbar, da diese die höchste Wertigkeit besitzen. Die folgende Tabelle listet die Operatoren sortiert von der höchsten zur niedrigsten Priorität auf. Operatoren, die in der gleichen Zeile stehen, sind gleichwertig und werden, wenn sie in einem Ausdruck zusammen benutzt werden, einfach von links nach rechts abgearbeitet.

Dazu Beispiele:	( )	Klammern
	- @	negatives Vorzeichen, Adresse
	* / MOD LSH RSH	Multiplikation, Division, Divisionsrest, Links-, Rechtsverschiebung
	+ -	Addition, Subtraktion
	= # <> > >= < <=	relationale Operatoren
	AND &	logisches, bitweises UND
	OR %	logisches, bitweises ODER
	XOR !	bitweises EXKLUSIV ODER

Nach dieser Übersicht wird das anfängliche Beispiel '4+5*3 als 4+(5*3)' bearbeitet, weil '*' eine höhere Priorität besitzt als '+'. Und wenn nun '(4+5)*3' gerechnet werden soll?

Dann muss man Klammern einfügen, um die normale Rangfolge abzuändern. Einige Beispiele zur Verdeutlichung:

Ausdruck	**Ergebnis**	**Rangfolge**
4/2*3	6	/,*
5<7	WAHR	<
43MOD7*2+19	21	MOD,*,+
-((4+2)/3)	-2	+,/,-

4.2 Ausdrücke

Ein arithmetischer Ausdruck besteht aus einer Anzahl von numerischen Konstanten, Variablen und Operatoren, die so geordnet sind, dass ein numerisches Ergebnis erzielt wird. Die Reihenfolge sieht so aus:

<Operand> <Operator> <Operand>

Wobei der <Operand> eine numerische Konstante, eine numerische VAR, ein Aufruf einer FUNC[43] oder ein anderer arithmetischer Ausdruck sein kann. Die ersten drei Möglichkeiten sind ziemlich einfach, die Letzte stellt aber ein Problem dar. Zur Verdeutlichung ein Beispiel anhand des ursprünglicher Ausdrucks '3*(4+(21/7)*2)':

Rang	**Ausdruck**	**Bearbeitung**	**vereinfacht**
Start	3*(4+(21/7)*2)	----	----
1	(21/7)	3	3*(4+3*2)
2	(21/7)*2	6	3*(4+6)
3	(4+(21/7)*2)	10	3*10
4	3*(4+(21/7)*2)	30	30

'Rang' ist die Reihenfolge, in welcher der Ausdruck abgearbeitet wird. 'Ausdruck' zeigt, welcher Teil des Gesamtausdrucks bearbeitet wird. Bearbeitung' gibt an, wie der Teil bearbeitet wird. 'Vereinfacht' stellt den Ausdruck so dar, wie er nach der Bearbeitung aussieht.

Beachtenswert, dass die Ausdrücke 2 bis 4 sich verändern. Das kommt daher, dass der 'Ausdruck als Operand' bereits bearbeitet ist und dadurch nur noch eine Zahl an dessen Stelle übrig bleibt, was unter 'vereinfacht' gezeigt wird.

Es folgen noch einige Beispiele, bei denen die in Kleinbuchstaben geschriebenen Wörter entweder Variablen oder Konstanten sind:

43) Kapitel IV, Abschnitt 6.2.3.

Ausdruck	Bearbeitungsfolge
'A*(dog+7)/3	+,*,/
564	(keine)
var & 7 MOD 3	MOD,&
ptr+ @ xyz	@,+

In Action! können Operanden aus verschiedenen Datentypen bestehen. Der Datentyp eines gemixten Operanden ergibt sich aus der nachfolgenden Tabelle. Er befindet sich am Schnittpunkt von Spalte und Zeile derjenigen Datentypen, die gemixt verwendet werden sollen:

	BYTE	INT	CARD
BYTE	BYTE	INT	CARD
INT	INT	INT	CARD
CARD	CARD	CARD	CARD

Anmerkung: Die Verwendung des negativen Vorzeichens '-' führt immer zu einem INT-Typ. Der Gebrauch des Adress-Operators '@' führt immer zu einem CARD-Typ.

Technische Anmerkung: Die Verwendung der Operanden '*', '/' oder 'MOD' resultiert in einem INT-Typ, weshalb die Verarbeitung von sehr großen CARD-Werten (>32767) nicht fehlerfrei funktioniert.

4.3 Einfache relationale Ausdrücke

Relationale Ausdrücke werden in bedingten Anweisungen verwendet, um zu überprüfen ob eine Anweisung bearbeitet werden soll[44]. Nicht vergessen, dass relationale Ausdrücke ausschließlich in bedingten Anweisungen (IF, WHILE, UNTIL) erlaubt sind.

In einem einfachen relationalen Ausdruck darf nur ein relationaler Operator verwendet werden. Überprüfungen, in denen mehrfache Bedingungen vorkommen, müssen deshalb anders gehandhabt werden. Im nachfolgenden Abschnitt werden sie unter ‘komplexe relationale Ausdrücke’ behandelt. Die allgemeine Form eines einfachen relationalen Ausdrucks lautet:

<arithmet. Ausdruck> <relationaler Operator> <arithmet. Ausdruck>

Es folgen einige Beispiele für zulässige einfache relationale Ausdrücke:

```
@ cat<=$22A7
var<>'y
5932#counter
(5&7)*8 >= (3*(cat+dog))
addr/$FF+(@ptr+offset) <> $F03D-ptr&offset
(5+4)*9 > ctr-1
```

44) Mehr dazu in Kapitel IV, Abschnitt 5.2.1.

4.4 Komplexe relationale Ausdrücke

Komplexe relationale Ausdrücke erlauben umfangreichere Überprüfungen durch den Einbau von Mehrfachtests. Soll etwas nur an Sonntagen im Juli erledigt werden, stellt sich die Frage, wie man das dem Computer beibringen kann. Er soll herausfinden, ob es Sonntag und ob es Juli ist. Action! ermöglicht diese Art von Mehrfachtests mit Hilfe der AND- und OR-Operatoren[45]. Der Compiler behandelt diese wie spezielle relationale Operatoren, die getestet werden, indem einfache relationale Operatoren zur Anwendung kommen. Die allgemeine Form lautet:

<rel. Ausdruck><spez. Operator><rel. Ausdr.> :<spez Op.><rel. Ausdr.>:

Anmerkung: Von dieser Form gibt es keine Ausnahmen. Versucht man etwas anderes, führt das in der Regel zum Compilerfehler 'Bad Expression' (falscher Ausdruck).

Die nachfolgende Wahrheitstabelle zeigt, was jeder der Operatoren im Falle einer Verknüpfung bewirkt. 'Ausdr. 1' und 'Ausdr. 2' sind einfache relationale Ausdrücke auf einer Seite des speziellen Operators; 'richtig' und 'falsch' sind mögliche Ergebnisse eines relationalen Tests.

rel. Ausdr.		**Ergebnisse**	
Ausdr.1	Ausdr.2	AND	OR
RICHTIG	RICHTIG	RICHTIG	RICHTIG
RICHTIG	FALSCH	FALSCH	RICHTIG
FALSCH	RICHTIG	FALSCH	RICHTIG
FALSCH	FALSCH	FALSCH	FALSCH

Anmerkung: Der Gebrauch von Klammern zum Ändern der Rangfolge bei der Bearbeitung ist zulässig. Beim Verzicht auf Klammern werden die Ausdrücke von links nach rechts abgearbeitet (siehe nachfolgende Beispiele).

Warnung: Zu dem Zeitpunkt der Erstellung dieses Handbuchs hatte der Compiler die Paare AND - & und OR - % als Synonyme behandelt; und sie werden auch auf gleiche Weise (bitweise) verarbeitet. Wer sich an die bisher dargelegten Regeln hält, wird keine Probleme bekommen. Also 'AND' und 'OR' bitte nur im relationalen Sinne und '&' und '%' nur im bitweisen Sinne benutzen.

Es folgen einige Beispiele für zulässige komplexe, relationale Ausdrücke:

```
cat<=5 AND dog<>13
(@ ptr+7)*3 # $60FF AND @ ptr <= $1FFF
x!$F0<>0 OR dog>=100
(8&cat)<10 OR @ ptr<>$0D
cat<>0 AND (dog>400 OR dog<-400)
ptr=$D456 OR ptr=$E000 OR ptr=$600
```

Und hier eine verwirrende Situation:

```
$F0 AND $0F
```

45) Kapitel IV, Abschnitt 4.1.3.

ist falsch, weil 'AND' als bitweiser Operator angesehen wird, der in einem arithmetischen Ausdruck verwendet wird. Im Gegensatz dazu ist

```
$F0<>0 AND $0F<>0
```

richtig, weil in diesem Falle 'AND' zwei einfache relationale Ausdrücke verknüpft und so zu einem speziellen Operator wird, der in einem komplexen, relationalen Ausdruck benutzt wird.

5 Anweisungen

Ein Computerprogramm wäre nutzlos, wenn es nicht selbsttätig auf Daten zugreifen könnte. Bisher können Variablen, Konstanten etc. deklariert werden, es gibt aber keine Möglichkeit der Manipulation. Anweisungen sind der aktive Teil jeder Computersprache, und Action! bildet da keine Ausnahme. Anweisungen übersetzen einen gewünschten Ablauf in eine Form, die der Computer verstehen und selbstständig ausführen kann. Das ist auch der Grund dafür, warum Anweisungen manchmal den ausführbaren Kommandos zugeordnet werden.

In Action! gibt es zwei Arten von Anweisungen: einfache und strukturierte. Einfache Anweisungen enthalten keine andere Anweisung als sich selbst. Strukturierte Anweisungen dagegen sind Zusammenstellungen von mehreren Anweisungen (entweder einfache oder strukturierte), zusammengefügt in einer bestimmten Reihenfolge. Strukturierte Anweisungen können in zwei Kategorien unterteilt werden:

- Bedingte Anweisungen
- Wiederholungsanweisungen

Sie werden im Abschnitt über die strukturierten Anweisungen diskutiert.

5.1 Einfache Anweisungen

Einfache Anweisungen erledigen nur eine Aufgabe. Sie sind die Bausteine eines jeden Programms. In Action! gibt es zwei einfache Anweisungen:

- Zuordnende Anweisungen (inkl. FUNC-Aufrufe)
- PROC-Aufrufe

PROC- und FUNC-Aufrufe werden im Abschnitt 6 erklärt. Als Nächstes folgen zuordnende Anweisungen. Außerdem gibt es noch zwei Schlüsselworte, die gleichfalls einfache Anweisungen sind.

EXIT	Abschnitt 5.2.3.2
RETURN	Abschnitte 6.1.2 und 6.2.2

Diese beiden werden nur in besonderen Konstruktionen benutzt und deshalb dort erläutert, wo ihre Anwendung gezeigt wird.

5.1.1 Zuordnende Anweisungen

Die zuordnende Anweisung wird benutzt, um einer VAR einen Wert zuzuweisen. Die gebräuchlichste Form ist:

<VAR>=<arithmetischer Ausdruck>

Anmerkung: <VAR> kann eine Variable vom elementaren Datentyp sein oder ein Array, ein Zeiger oder eine Registerangabe.

Hinweis: Der Ausdruck muss arithmetisch sein. Versucht man einen relationalen Ausdruck zu gebrauchen, führt das zu einem Fehler, weil der Compiler einen numerischen Wert nicht der Auswertung eines relationalen Ausdrucks zuordnen kann.

Der zuweisende Operator ist '='. Er teilt dem Computer mit, dass der gegebenen VAR ein neuer Wert zugewiesen werden soll. Das darf nicht mit dem relationalen '=' verwechselt werden. Obwohl es beides mal das gleiche Zeichen ist, liest es der Compiler unterschiedlich, je nach dem Zusammenhang, in dem es benutzt wird.

Die folgenden Beispiele verdeutlichen die zuordnende Anweisung. Sie werden einen Abschnitt zur Deklaration von Variablen den Beispielen vorangestellt finden. Das ist notwendig, weil einige der Beispiele zeigen, was passiert, wenn Typen gemixt werden (z.B. wenn Variable und der ihr zugewiesene Wert nicht vom gleichen Datentyp sind).

```
BYTE b1,b2,b3,b4
INT i1
CARD c1
```

b3='D schreibt die ATASCII-Code-Zahl für 'D' in das Byte, das für 'b3' reserviert wurde.

b4=$44 schreibt die Hex-Zahl $44 in das für die BYTE-Variable 'b4' reservierte Byte. $44 ist im ATASCII-Code das 'D', weswegen 'b3' und 'b4' jetzt den gleichen Inhalt haben.

b1=b4+16 addiert 16 zum numerischen Wert von 'b4' und schreibt das Ergebnis in das für 'b1' reservierte Byte.

c1=23439-$07D8 setzt den Wert 21431 ($53B7) in die zwei für 'c1' reservierten Bytes.

i1=c1*(-1) schreibt den Wert -21431 ($AC49) in die zwei für 'i1' reservierten Bytes.

b2=i1 schreibt den Wert $49 (73) in das für 'b2' reservierte Byte. Beachten Sie, dass der Computer das LSB von 'i1'nimmt und in 'b2' hineinschreibt. Das MSB von 'i1' ist $AC, das LSB ist $49.

b2=b2+1 addiert 1 zum momentanen Wert von 'b2' und schreibt die Summe zurück in 'b2'. 'b2' enthält jetzt $4A (74).

Die Form des letzten Beispiels lautet:

<Variable>=<Variable><Operator><Operand>

Da diese Form oft verwendet wird, bietet Action! folgende Kurzform dafür an:

<Variable>==<Operator><Operand>

Der Operator muss entweder arithmetisch oder bitweise arbeiten. Der Operand muss ein arithmetischer Ausdruck sein. Ein paar Beispiele für diese Kurzform:

b2==+1	entspricht	b2=b2+1
b2==-b1	entspricht	b2=b2-b1
b2==& $0F	entspricht	b2=b2 & $0F
b2==LSH (5+3)	entspricht	b2=b2 LSH (5+3)

Diese Kurzform spart doch einiges an Zeit beim Eingeben und generiert manchmal sogar einen besseren Maschinencode.

5.2 Strukturierte Anweisungen

Wenn nur einfache Anweisungen zugelassen wären, würde man beim Programmieren manchmal ganz schönen Beschränkungen unterliegen.

So wäre der einzige Weg, eine Gruppe von Anweisungen in einer bestimmten Anzahl wiederholen zu lassen, sie in der gewünschten Reihenfolge und Anzahl einzugeben. Will man also 10 Anweisungen 10x wiederholen, müssten sie 100 Mal eingegeben werden!

Es wäre auch nicht möglich, eine Gruppe von Anweisungen bedingt ausführen zu lassen, sondern nur dann, wenn festgelegte Prüfungen zuvor erfolgreich gewesen wären.
Zur Lösung dieser und anderer Probleme gibt es die strukturierten Anweisungen, die sich in zwei Kategorien unterteilen lassen: Bedingte und Wiederholungsanweisungen. Beide Arten werden für sich betrachtet.

5.2.1 Bedingte Ausführung

Bedingungen ermöglichen es, einen Ausdruck zu überprüfen und in Abhängigkeit vom Ergebnis verschiedene Anweisungen ausführen zu lassen. Da der Ausdruck die bedingte Ausführung überwacht, wird er als bedingter Ausdruck bezeichnet.

Drei Anweisungen in Action! erlauben eine bedingte Ausführung:

IF WHILE UNTIL

Die Schleifenanweisungen WHILE und UNTIL werden später behandelt, aber IF wird im Anschluss an die Regeln für bedingte Ausdrücke erklärt.

5.2.1.1 Bedingte Ausdrücke

Da ein bedingter Ausdruck überprüft wird, kann es nur zwei mögliche Ergebnisse geben - richtig oder falsch. Bedingte Ausdrücke sind keine neue Art von Ausdrücken, sondern einfach nur relationale oder arithmetische Ausdrücke. Unterschiedlich ist nur die Ausführung.

Die nachfolgende Tabelle zeigt, welche Art von Interpretation vorliegt, abhängig von der Art des Ausdrucks:

Ausdrucksart	normales Ergebnis	bedingtes Ergebnis
arithmetisch	null (0) nicht null	FALSCH RICHTIG
relational	FALSCH RICHTIG	FALSCH RICHTIG

5.2.1.2 IF-Anweisungen

Die IF-Anweisung in Action! entspricht dem 'if' (wenn) im Englischen.

Beispiel: "Wenn (IF) ich 250 € oder mehr habe, kaufe ich mir noch eine Floppy 1050."

In Action! könnte diese Anweisung so aussehen:

```
BYTE Geld,
     F1050=[250]
     XF551=[215]
     IndusGT=[200]
     F810=[180]
IF Geld>=250 THEN
     kaufe(F1050,Geld)
FI
```

Anmerkung: kaufe(F1050,Geld) ist ein sogenannter Aufruf einer PROC und wird in Abschnitt 6.1.3 behandelt.

Aus dem vorigen Beispiel wird die grundsätzliche Form der IF-Anweisung erkennbar:

```
IF   <bedingter Ausdruck> THEN
     <Anweisung(en)>
FI
```

'FI' ist die Umkehrung von 'IF' und ein Schlüsselwort für den Compiler, an dem er erkennt, dass die IF-Anweisung zu Ende ist. Da mit IF eine ganze Reihe von Anweisungen bearbeitet werden kann, wird das 'FI' zum Beenden benötigt. Ohne dieses Schlüsselwort wüsste der Compiler nicht, wie viele Anweisungen nach dem THEN durch die IF-Anweisung verarbeitet werden sollen.

Zusätzlich zur Grundform gibt es noch zwei andere Varianten, ELSE und ELSEIF. Im Englischen gibt es diese Optionen auch, deshalb hier ein paar vergleichbare Beispiele:

"Wenn ich 250 € oder mehr habe, kaufe ich mir eine XF551, andernfalls eine F1050."

In Action! wird daraus:

```
IF Geld>=250 THEN
    kaufe(XF551,Geld)
ELSE
    kaufe(F1050,Geld)
FI
```

ELSEIF ist noch etwas anders:

"Wenn ich 250 € oder mehr habe, kaufe ich die XF551.
Andernfalls, wenn (ELSEIF) ich zwischen 215 € und 250€ habe, kaufe ich die Indus GT.
Andernfalls, wenn (ELSEIF) ich zwischen 200 € und 215 € habe, kaufe ich mir die F1050, andernfalls die F810."

In Action! wird daraus:

```
IF Geld>=250 THEN
    kaufe(XF551,Geld)
  ELSEIF Geld>=215 THEN
    kaufe(IndusGT,Geld)
  ELSEIF Geld>=200 THEN
    kaufe(F1050,Geld)
  ELSE
    kaufe(F810,Geld)
FI
```

Die Prüfung "Geld>=215 AND Geld<250" ist nicht notwendig, weil der Computer die Liste sequenziell von oben nach unten abarbeitet. Ist eine Bedingung erfüllt, wird die von ihr gesteuerte Anweisung ausgeführt und der Rest der IF-Anweisungen übersprungen. Führt also der Computer die zu "Geld>=215" gehörende Anweisung aus, ist klar, dass weniger als 250€ vorhanden sind, weil der Computer die vorherige Bedingung geprüft und mit falsch bewertet hat.

Die Option ELSEIF ist sehr nützlich, wenn eine Variable auf mehrere Bedingungen hin geprüft werden soll, zu denen dann jeweils eine andere Programmausführung gehört.

5.2.2 Leer-Anweisungen

Eine Leer-Anweisung wird eingesetzt, um nichts zu tun. Nachdem bisher Anweisungen vorgestellt wurden, die etwas bewirken nun Anweisungen, die einfach nichts bewirken? Tatsächlich gibt es einige sehr nützliche Anwendungen für das Nichts: die zeitliche Abstimmung (Timing) von Schleifen sowie Abarbeitung von ELSEIF-Fällen. Da Schleifen bisher noch nicht erläutert wurden, wird einfach angenommen, dass es sich dabei um Zeitverzögerungen handelt, z.B. Pausen bei der Bildschirmausgabe[46].

Das nächste Beispiel verdeutlicht die Verwendung von Leer-Anweisungen mit ELSEIF.

Vorgaben: Es soll ein Programm entstehen, mit dem Börsenmakler durch die eingebauten Anweisungen Informationen über bestimmte Aktien gewinnen können. Diese lauten: KAUFEN, FALLEN?, SUCHEN, BEENDEN, VERKAUFEN und ANSTEIGEN?, doch für SUCHEN wurde noch kein Unterprogramm geschrieben. Man braucht nur den ersten Buchstaben jeder Anweisung abzufragen, also nach K, F, S, B, V, A zu suchen. Die Funktion SUCHEN ist aber noch nicht im Programm enthalten. Was passiert also, wenn jemand 'S' eingibt? Ganz einfach, nichts! Erst wenn später das Unterprogramm SUCHEN eingefügt wird, kann es auch ausgeführt werden. Bis dahin bleibt eine Leer-Anweisung im Programm stehen. Der entsprechende Programmteil kann dann etwa so aussehen:

```
IF chr='K THEN
    dokaufen()
  ELSEIF chr='F THEN
    dofallen()
  ELSEIF chr='S THEN
    ;*** hier steht die Leer-Anweisung
  ELSEIF chr='B
```

46) Beispiel dazu in Kapitel IV, Abschnitt 5.2.4.1.

```
          dobeenden()
        ELSEIF chr='V THEN
          doverkaufen()
        ELSEIF chr='A THEN
          doansteigen()
        ELSE
          doFehler() ;*** kein Befehl traf zu
      FI
```

Alle 'do---'s sind PROCs, die die gewünschten Anweisungen ausführen. Betrachtet man die Zeile mit "chr='S": Es passiert nichts. Das ist die Leer-Anweisung. Sobald die Prozedur SUCHEN fertig programmiert ist, braucht man nur noch 'dosuchen()' anstelle der Leer-Anweisung einsetzen und die PROC zum Programm hinzufügen. Damit ist das Programm dann vollständig und kann benutzt werden.

5.2.3 Schleifen

Schleifen werden für Wiederholungen vor allem bei Anweisungen eingesetzt. Soll der Bildschirm vollkommen mit Sternchen (*) beschrieben werden, kann entweder jedes Sternchen einzeln per Anweisung oder mit einer Schleife auf den Bildschirm gebracht werden. Man muss der Schleife nur mitteilen, wie oft ein Sternchen ausgegeben werden soll und sie wird es ausführen (natürlich nur, wenn die Form der Anweisung korrekt ist).

Es gibt zwei Möglichkeiten einer Schleife mitzuteilen, wie oft etwas ausgeführt werden soll. Es kann eine bestimmte Anzahl vorgeben oder ein bedingter Ausdruck verwendet werden, von dessen Ergebnis die Anzahl der Schleifendurchläufe abhängt. Die FOR-Anweisung benutzt die erste Methode, WHILE und UNTIL benutzen die zweite.

Was passiert, wenn der Schleife nicht mitgeteilt wird, wie oft sie ausgeführt werden soll? Was passiert, wenn der bedingte Ausdruck niemals zu einem Ergebnis führt, das die Schleife beendet? Das führt in eine sogenannte 'Endlosschleife'. Aus dieser führt nur noch ein Weg heraus: Drücken von <SYSTEM RESET>!!!

Action! behandelt Schleifen folgendermaßen. Es gibt eine einfache Schleife, die für sich allein verwendet unendlich läuft. Dazu gibt es die Schleifen steuernde Anweisungen (FOR, WHILE, UNTIL), welche die Anzahl der Ausführungen einer an sich unendlichen Schleife begrenzen. Zuerst wird die Struktur der einfachen Schleife erläutert, danach folgt die intensive Betrachtung der Steuerungsanweisungen für Schleifen.

5.2.3.1 DO und OD

'DO' und 'OD' markieren Anfang und Ende einer einfachen Schleife. Alles, was dazwischen steht, ist Bestandteil derselben. Wie schon erwähnt, ist eine Schleife ohne steuernde Anweisung eine endlose Schleife, aus der man nur gewaltsam ausbrechen kann. Das folgende Beispiel demonstriert die DO-OD-Schleife. Die Anweisungen 'PROC' und 'RETURN' sind für den Compiler wichtig und nicht Gegenstand der Betrachtung. Durch sie wird das Programm erst lauffähig, was in Abschnitt 6 eingehend besprochen wird.

Beispiel:

```
PROC zweimal()
  CARD i=[0],j[47]

  DO                          ; Start der DO-OD-Schleife
    i==+1                     ; addiere 1 zu 'i'
    j=i*2                     ; definiere 'j' als i*2
    PrintC(i)                 ; *** Siehe nachfolgende
    Print(" mal 2 ergibt ")   ;     ANMERKUNG
    PrintCE(j)
  OD                          ; Ende der DO-OD-Schleife
RETURN
```

Anmerkungen:
Die in Groß- und Kleinbuchstaben geschriebenen Wörter (PrintC, Print, PrintCE) in vorigem Beispiel sind FUNCs und PROCs aus der Action!-Library[48]. In diesem Beispiel und im Rest des Kapitels werden immer wieder solche Library-Routinen verwendet, um die Beispiele eingängiger zu gestalten.

Bildschirmausgabe des Beispiels:

```
1 mal 2 ergibt 2
2 mal 2 ergibt 4
3 mal 2 ergibt 6
4 mal 2 ergibt 8
5 mal 2 ergibt 10
6 mal 2 ergibt 12
(usw.)
```

Die Bildschirmausgabe läuft unendlich weiter, es sei denn, man drückt <SYSTEM RESET>. Für sich allein ist eine DO-OD-Schleife ziemlich nutzlos. Wendet man sie aber in Verbindung mit den die Schleifen steuernden Anweisungen FOR, WHILE und UNTIL an, wird sie zu einer sehr nützlichen Anweisung.

Anmerkungen: Mit der <BREAK>-Taste kann die Endlosschleife im ersten Beispiel auch beendet werden. Das geht deshalb, weil in der Schleife viele Ein-/Ausgaben (I/O) vorkommen. <BREAK> funktioniert in Action!, wenn I/Os ausgeführt werden[49].

Bitte immer daran denken, dass eine 'DO-OD-Schleife' auch von 'DO-OD' eingerahmt sein muss, damit sie funktioniert.

5.2.3.2 EXIT-Anweisung

Mithilfe der EXIT-Anweisung kommt man elegant aus jeder Schleife heraus. Diese Anweisung veranlasst die Programmausführung bei der Anweisung fortzufahren, die auf das nächste 'OD' folgt. Einige Beispiele dazu:

---

47) CARD i=[0] wir nur einmalig beim Kompilieren auf 0 gesetzt. Siehe auch Abschnitt 6.3.
48) Mehr darüber in Kapitel VI.
49) Kapitel III, Abschnitt 3 und Kapitel VI, Abschnitt 7.2.

Beispiel 1

```
PROC zweimal()

  CARD i=[0],j
  DO                      ; Start der DO-OD-Schleife
    i==+1                 ; addiere 1 zu 'i'
    j=i*2                 ; definiere 'j' als i*2
    PrintC(i)
    Print(" mal 2 ergibt ")
    EXIT                  ; die EXIT-Anweisung
    PrintCE(j)
  OD                      ; Ende der DO-OD-Schleife
  ; *** hier geht's nach EXIT weiter
  PrintE("Ende der Tabelle")
RETURN
```

Bildschirmausgabe Beispiel 1

```
1 mal 2 ergibt Ende der Tabelle
```

Wie die Ausgabe zeigt, wird die Anweisung 'PrintCE(j)' nicht mehr ausgeführt. Die EXIT-Anweisung zwingt das Programm, zur Anweisung 'PrintE("Ende der Tabelle")' zu springen. Allein ist die Verwendung von EXIT nicht sehr sinnvoll. Aber in Verbindung mit einer IF-Anweisung (um z.B. das Verlassen der Schleife von Bedingungen abhängig machen) kann EXIT sehr nützlich sein, wie das nächste Programm zeigt.

Beispiel 2

```
PROC zweimal()

  CARD i=[0],j

  DO                        ; Start der DO-OD-Schleife
    IF i=14 THEN
      EXIT                  ; EXIT in einer IF-Bedingung
    FI
    i==+1
    j=i*2
    PrintC(i)
    Print(" mal 2 ergibt ")
    PrintCE(j)
  OD                        ; Ende der DO-OD-Schleife
  ; *** hier geht's weiter, wenn i=14
  PrintE("Ende der Tabelle")
RETURN
```

Bildschirmausgabe Beispiel 2

```
1 mal 2 ergibt 2
2 mal 2 ergibt 4
```

```
3 mal 2 ergibt 6
4 mal 2 ergibt 8
5 mal 2 ergibt 10
6 mal 2 ergibt 12
7 mal 2 ergibt 14
8 mal 2 ergibt 16
9 mal 2 ergibt 18
10 mal 2 ergibt 20
11 mal 2 ergibt 22
12 mal 2 ergibt 24
13 mal 2 ergibt 26
14 mal 2 ergibt 28
Ende der Tabelle
```

Diese Anwendung verwandelt eine unendliche Schleife in eine endliche. EXIT kann also die Ausführung einer Schleife steuern, wird selbst aber nicht als strukturierte, Schleifen steuernde Anweisung betrachtet, weil es allein nicht sinnvoll angewendet werden kann. Deshalb macht EXIT nur Sinn, wenn es in Verbindung mit einer strukturierten IF-Anweisung zur Anwendung gelangt.

5.2.4 Schleifensteuerung

Action! verfügt über drei strukturierte Anweisungen zur Steuerung der einfachen DO-OD-Schleife:

- FOR
- WHILE
- UNTIL

Durch sie wird die Anzahl der Ausführungen begrenzt und aus einer endlosen so eine endliche Schleife. Dank der steuerbaren Schleifen wird die Strafarbeit „Schreibe 1000-mal Action! ist die schnellste Hochsprache für XL/XE!“ mithilfe des Computers fast schon zum Vergnügen.

Als Nächstes wird jede einzelne Steuerungsanweisung genau beleuchtet und dann folgt die Erläuterung einer gemeinsamen Eigenschaft aller Action!-Anweisungen: der Verknüpfung.

5.2.4.1 FOR-Anweisung

Die FOR-Anweisung wird dazu verwendet, eine Schleife in einer vorgegebenen Anzahl zu wiederholen. Sie braucht dazu eine eigene Variable, allgemein als Zähler (Counter) bezeichnet. In den Beispielen wird der Zähler als 'ctr' benannt. Man kann ihn aber auch beliebig anders benennen. Die Form der FOR-Anweisung lautet:

```
FOR     <counter>=<initial value> TO <final value> {STEP <inc> }

        <DO-OD-Schleife>
```

wobei

<counter>	als Variable die gegebene Anzahl an Wiederholungen enthält.
<initial value>	der Startwert des Zählers ist

<final value> der zu erreichende Endwert des Zählers ist
<inc> der Wert ist, der bei jeder Wiederholung zum Zähler zugerechnet wird. STEP <inc> ist optional.
<DO-OD-Schleife> die unendliche DO-OD-Schleife ist.

Anstelle weiterer Erklärungsversuche jetzt einige sich mehr oder weniger selbst erklärende Beispiele. Bitte die Ausgabe der Beispiele beachten.

Beispiel 1

```
PROC Hallo()
  BYTE ctr              ; Zaehler für die FOR-Schleife
  FOR ctr=1 TO 3        ; Kein 'STEP' bedeutet stets
    DO                  ; Erhoehung um 1.
      PrintE("Hallo User!")
    OD
RETURN
```

Ausgabe:
```
Hallo User!
Hallo User!
Hallo User!
```

Beispiel 2

```
PROC Gerade_Zahl()

BYTE ctr                 ; Zaehler für die FOR-Schleife

FOR ctr=0 TO 16 STEP 2   ; diesmal mit 'STEP'
  DO
    PrintB(ctr)
    Print(" ")
  OD
RETURN
```

Ausgabe:
```
0 2 4 6 8 10 12 14 16
```

In der allgemeinen Form der FOR-Anweisung findet man nirgendwo etwas über die Verwendung von numerischen Variablen als <initial value>, <final value> oder <inc>. Dies ist zulässig und erlaubt es, FOR-Schleifen in variabler Anzahl zu wiederholen.

Durch Änderung der VAR-Werte für <initial value>, <final value> oder <inc> innerhalb der Schleife lässt sich die Anzahl der Schleifendurchläufe nicht verändern. Die aktuellen Werte der VARs werden nämlich bei Beginn der Schleife als Konstante übernommen.

Wird der Wert von <counter> in der Schleife verändert, ändert sich auch die Anzahl der Durchläufe, weil es eine Variable ist, die innerhalb der Schleife steht. Der Wert von <counter> lässt sich innerhalb der Schleife deshalb verändern, weil die FOR-Anweisung selbst den Wert von <counter> bei jedem Schleifendurchlauf um den STEP-Wert ändern muss. Ein Beispiel dazu:

Beispiel 3

```
PROC Aenderschleife()

BYTE ctr,
     start=[1],
     end=[50]

FOR ctr=start TO end
  DO
    start=100  ; veraendert nicht die Anzahl der Wiederholungen
    end=10     ; veraendert nicht die Anzahl der Wiederholungen
    PrintBE(ctr)
    ctr==*2     ; veraendert die Anzahl der Wiederholungen
  OD
RETURN
```

Ausgabe:
```
1
3
7
15
31
```

Die nachfolgende Tabelle zeigt, was in jedem Schleifendurchlauf passiert.

Nr	neu ctr	Print	ctr==*2
1	–	1	2
2	3	3	6
3	7	7	14
4	15	15	30
5	31	31	62

'Nr' gibt die Zahl des Durchlaufs an. 'neu ctr' gibt den durch die FOR-Schleife veränderten Wert des Zählers an. 'Print' zeigt an, was auf dem Bildschirm ausgegeben wird. Und 'ctr==*2' als zuordnende Anweisung verändert den Wert des Schleifenzählers.

Nach dem fünften Durchlauf hat der Zähler den Wert 62 erreicht. Da dieser schon größer als der <final value> (50)ist, wird die Ausführung der Schleife schon nach 5 und nicht erst nach 50 Durchläufen beendet. Den Zähler innerhalb seiner eigenen Schleife zu verändern, kann manchmal recht nützlich sein.

Wie in Abschnitt 5.2.2 angesprochen nun eine Zeitschleife:

```
BYTE ctr

FOR ctr=1 to 250
   DO
```

```
   ; *** dies ist eine Leer-Anweisung
   OD
```

Diese Warteschleife wird für ein sauberes Timing benötigt.

Anmerkung: Schreibt man eine FOR-Schleife, deren Zähler die Grenzen seines Datentyps erreicht, führt das zu einer unendlichen Schleife. Der Zähler kann nicht über diese Grenze hinaus zählen und daher die Schleife nicht beenden. Bei BYTE ist das bei 'FOR ctr=0 TO 255' und bei CARD bei 'FOR ctr=0 TO 65535' der Fall. Die Schleife wird nämlich erst dann beendet, wenn der <counter> größer als der <final value> ist.

5.2.4.2 WHILE-Anweisung

Die WHILE-Anweisung (ebenso die UNTIL-Anweisung) ist hilfreich, wenn eine Schleife eine vorher nicht festgelegte Anzahl an Durchläufen ausführen soll. WHILE erlaubt eine Schleife solange ausführen zu lassen, wie eine festgelegte Bedingung 'richtig' ist. Sie hat die allgemeine Form:

WHILE <cond exp> <DO-OD-Schleife>

wobei

<cond exp>	der bedingte Ausdruck ist, der steuert.
<DO-OD-Schleife>	die unendliche DO-OD-Schleife ist.

Sind die Vorgaben des bedingten Ausdrucks bereits vor dem Start der Schleife erfüllt, wird sie gar nicht erst ausgeführt. Bei UNTIL ist das nicht so, wie man später sehen wird. Nun einige Beispiele mit WHILE.

Beispiel 1

```
PROC Fakultäten()
; *** Diese Prozedur gibt auf dem Bildschirm
; Fakultäten aus bis maximal 6000
CARD fact=[1],          ; Fakultaet von 'num'
     num=[1],           ; Zaehler
     amt=[6000]         ; obere Testgrenze

Print("Fakultaeten kleiner als ")
PrintC(amt)             ; gibt obere Grenze aus
PrintE(":")             ; gibt Doppelpunkt und <RETURN> aus
PutE()                  ; gibt ein <RETURN> aus
WHILE fact*num < amt    ; teste naechste Fakultaet
DO                      ; Start der Schleife
  fact==*num
  PrintC(num)           ; gibt die Zahl aus
  Print(" Fakultät ist ")
  PrintCE(fact)         ; gibt die Fakultaet zur
                        ; Zahl aus
  num==+1               ; erhoeht die Zahl
OD                      ; Ende der Schleife
RETURN                  ; Ende der PROC Fakultaeten
```

Ausgabe:
```
Fakultaeten kleiner als 6000:
1 Fakultaet ist 1
2 Fakultaet ist 2
3 Fakultaet ist 6
4 Fakultaet ist 24
5 Fakultaet ist 120
6 Fakultaet ist 720
7 Fakultaet ist 5040
```

Anmerkung: Der Compiler prüft nicht, ob ein Überlauf (Overflow)[50] stattfindet. Fakultäten größer als 65.535 erzeugen einen Überschlag: Die Ausgabe fängt wieder bei null an. Der Grund ist das für eine CARD zulässige Maximum von 65535. Die Testobergrenze von z.B. 66.000 wird als 66000-65535=464 ausgegeben, weil der Rechner bei CARDs nur bis 65535 zählen kann und dann wieder bei null anfängt.

Beispiel 2

```
PROC Ratespiel()
; *** Diese Prozedur spielt mit dem Anwender ein
; Ratespiel, das eine WHILE-Schleife verwendet
BYTE num,                ; die zu ratende Zahl wird auf
guess=[200]              ; einen unzulaessigen Wert gesetzt

PrintE("Willkommen beim Ratespiel! Ich ")
PrintE("denke mir eine Zahl zwischen 0 und 100.")
num=Rand(101)            ; ermittelt die zu ratende Zahl
WHILE guess<>num
DO                       ; Start der Schleife
  Print("Welche Zahl ? ")
  guess=InputB()         ; geratene Zahl
  IF guess<num THEN      ; Zahl zu klein
    PrintE("Zu klein, nochmal!")
  ELSEIF guess>num THEN  ; Zahl zu gross
    PrintE("Zu gross, nochmal!")
  ELSE                   ; richtige Zahl
    PrintE("Glueckwunsch!!!")
    PrintE("Du hast's geschafft!!!")
  FI                     ; Ende der Tests
OD                       ; Ende der Schleife
RETURN                   ; Ende der PROC Ratespiel
```

Ausgabe:
```
Willkommen zum Ratespiel!
Ich denke mir eine Zahl zwischen 0 und 100.
Welche Zahl ? 50
Zu klein, nochmal!
Welche Zahl ? 60
Zu gross, nochmal!
Welche Zahl ? 55
Zu klein, nochmal!
```

50) Mehr dazu im Kapitel VI.

```
         Welche Zahl ? 57
         Glueckwunsch!!!
         Du hast's geschafft!!!
```

Bemerkenswert, wie leistungsfähig manipulierende Bedingungen wie IF innerhalb einer Schleife sein können. Dadurch wird dem Computer bei jedem Schleifendurchlauf zur Verarbeitung eine Vielzahl an möglichen Ausgaben bereitgestellt.

5.2.4.3 UNTIL-Anweisung

Wie schon im letzten Abschnitt erwähnt, kann eine WHILE-Schleife auch null Mal ausgeführt werden, wenn der bedingte Ausdruck bereits beim Start der Schleife erfüllt war. Die allgemeine Form der UNTIL-Anweisung ist so gestaltet, dass diese Schleife mindestens einmal durchlaufen wird. Man wird gleich sehen, warum:

```
     DO
       <statement>
       :
       :
       UNTIL <cond exp>
     OD
```

Das sieht eigentlich nach einer einfachen DO-OD-Schleife aus, bis auf den Ausdruck, der unmittelbar vor dem 'OD' steht. Dieses UNTIL steuert die unendliche Schleife über das Ergebnis des bedingten Ausdrucks. Wenn <cond exp> 'wahr' ist, dann wird die Schleife beendet und mit der Anweisung nach dem 'OD' fortgefahren. Ist <cond exp> falsch, wird zu 'DO' zurückgesprungen und die Schleife erneut durchlaufen. Das folgende Beispiel wird Klarheit darüber schaffen:

```
PROC Ratespiel()
; *** Diese PROC spielt mit dem Anwender ein
; Ratespiel, das eine UNTIL-Schleife verwendet

  BYTE num,                  ; die zu erratende Zahl
       guess                 ; die geratene Zahl
  PrintE("Willkommen beim Ratespiel! Ich ")
  PrintE("denke mir eine Zahl zwischen 0 und 100.")
  num=Rand(101)              ; gibt die zu ratende Zahl
  DO                         ; Start der Schleife
    Print("Welche Zahl ? ")
    guess=InputB()           ; geratene Zahl
    IF guess<num THEN        ; Zahl zu klein
      PrintE("Zu klein, nochmal!")
    ELSEIF guess>num THEN    ; Zahl zu gross
      PrintE("Zu groß, nochmal!")
    ELSE                     ; richtige Zahl
      PrintE("Glueckwunsch!!!")
      PrintE("Du hast's geschafft!!!")
    FI                       ; Ende der Tests
    UNTIL guess=num          ; Schleifensteuerung
  OD                         ; Ende der Schleife
RETURN                       ; Ende der PROC Ratespiel
```

Ausgabe:
```
Willkommen zum Ratespiel!
Ich denke mir eine Zahl zwischen 0 und 100.
Welche Zahl ? 50
Zu klein, nochmal!
Welche Zahl ? 60
Zu gross, nochmal!
Welche Zahl ? 55
Zu klein, nochmal!
Welche Zahl ? 57
Glueckwunsch!!!
Du hast's geschafft!!!
```

Das Beispiel ist entspricht dem zur WHILE-Schleife, nur diesmal unter Verwendung einer UNTIL-Schleife. Aber im Gegensatz zur WHILE-Schleife wird 'guess' in der Variablendeklaration nicht initialisiert. Das ist möglich, weil der bedingte Ausdruck 'guess=num' erst dann geprüft werden kann, wenn für 'guess' eine Eingabe erfolgt ist. Der Vorteil der UNTIL-Schleife ist eben die Steuerung durch den bedingten Ausdruck am Ende der Schleife. WHILE dagegen verlangt bereits einen bedingten Ausdruck beim Start der Schleife, weshalb 'guess' dann bereits einen Wert haben muss.

5.2.5 Verschachteln von strukturierten Anweisungen

Wie schon im Überblick zu Anweisungen angesprochen, werden strukturierte Anweisungen aus anderen Anweisungen plus Steuerungsinformationen für deren Ausführung zusammengesetzt. Die Anweisungen innerhalb einer strukturierten Anweisung können entweder einfache Anweisungen oder andere strukturierte Anweisungen sein.

Das Einbauen einer strukturierten Anweisung in eine andere wird Verschachtelung genannt. In 5.2.4.2 (WHILE) und 5.2.4.3 (UNTIL) waren Beispiele zu sehen, wie eine IF-Anweisung in eine WHILE- bzw. UNTIL-Schleife eingebaut wird. Diese Art der Verschachtelung dürfte damit klar sein. Es folgt eine Erläuterung der mehrfachen Verschachtelung von gleichartig strukturierten Anweisungen (IFs in IFs, FORs in FORs usw.).

Wird eine IF-Anweisung in eine andere IF-Anweisung eingesetzt, kann Verwirrung darüber entstehen, welches ELSE zu welchem IF gehört; vor allem wenn mehrere IFs miteinander verschachtelt sind. Der Compiler vermeidet Verwirrung dadurch, dass er immer IF-FI-Paare zur Kompilierung sucht. Ein FI wird stets dem zuletzt gesetzten IF zugeordnet, dem noch kein FI zugehörig ist.

Beispiel:

```
┌ IF <expA> THEN
|   ┌ IF <expB> THEN
|   |     <statements>
|   | ELSEIF <expC> THEN      ; *** ELSEIF zu IF <expB>
|   |     ┌ IF <expD> THEN
|   |     | <statements>
|   |     | ELSE              ; *** ELSE zu IF <expD>
|   |     | <statements>
|   |     └ FI                ; *** Ende von IF <expD>
|   └ FI                      ; *** Ende von IF <expB>
| ELSEIF <expE> THEN          ; *** ELSEIF zu <expA>
|     <statements>
| ELSE                        ; *** ELSE zu IF <expA>
|     <statements>
└ FI                          ; *** Ende von IF <expA>
```

Die senkrechten Linien zeigen die IF-FI-Paare an. Die Kommentare erläutern, welche IF-Anweisungen zu welchen FI- oder ELSEIF-Teilen gehören. Darüber hinaus gibt das Einrücken der Zeilen die jeweiligen Stufen an.

Das nachfolgende Programm enthält verschachtelte FORs. Das Beispiel gibt die Multiplikationstabelle bis zehn mal zehn aus.

```
PROC timestable()
; *** Die PROC gibt die Multiplikationstabelle bis 10 * 10 aus.

BYTE c1,                 ; Zaehler der aeusseren FOR-Schleife
     c2                  ; Zaehler der inneren FOR-Schleife

FOR c1=1 TO 10                ; Steuern der aeusseren Schleife
  DO                           ; Start der aeusseren Schleife
    IF c1<10 THEN              ; 1-stellige Zahlen brauchen
      Print(" ")               ; in der 1. Spalte
    FI                         ; 1 Leerzeichen
    PrintB(c1)                 ; erste Zahl in Spalte ausgeben
    FOR c2=2 TO 10             ; Steuern der inneren Schleife
      DO                       ; Start der inneren Schleife
        IF c1*c2 < 10 THEN     ; einstellige Zahlen
          Print("   ")         ; brauchen 3 Leerzeichen
        ELSEIF c1*c2<100 THEN  ; 2-stellige Zahlen
          Print("  ")          ; brauchen 2 Leerzeichen
        ELSE                   ; 3-stellige Zahlen
          Print(" ")           ; brauchen nur 1 Leerzeichen
        FI                     ; Ende der Leerstellen
        PrintB(c1*c2)          ; Ergebnis ausgeben
      OD                       ; innere Schleife beenden
    PutE()                     ; Return ausgeben
  OD                           ; Ende der aeusseren Schleife
RETURN                         ; Ende der PROC timestable
```

Ausgabe:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Wie das obige Beispiel zeigt, ist eine Verschachtelung durchaus sinnvoll nutzbar, wenn man genau weiß wofür. Glücklicherweise braucht das "wissen wofür" kaum Zeit, weil die Konzeption der Verschachtelung für alle strukturierten Anweisungen gleich ist. Hat man erst einmal verstanden, wie die Verschachtelung für eine Anweisung eingesetzt werden kann, kann man sie auf alle Anweisungen anwenden.

6 Prozeduren und Funktionen

PROCs und FUNCs werden gebraucht, um ein Programm besser lesbar und benutzbar zu gestalten. Im Grunde genommen ist alles, was man auf die eine oder andere Art tut, eine PROC oder FUNC. Dazu die folgende Tabelle:

Prozeduren	**Funktionen**
Auto waschen	Kontostand prüfen
Geschirr spülen	Telefonnummer 'raussuchen
zur Arbeit fahren	Radio einschalten

Was macht diese PROCs und FUNCs aus? Nun, für beide gilt:

- Es ist eine Gruppe von zusammenhängenden Tätigkeiten, die zur Erledigung einer Aufgabe ausgeführt werden.

- Es gibt eine logische Reihenfolge, in der die Schritte erfolgen müssen.

Das Geschirr abtrocknen, bevor es gespült wird, ergäbe keine logische Reihenfolge; ebenso wie das Ausziehen der linken Socke keine Beziehung zum Vorgang "Geschirrspülen" hat. So etwas weiß man aus Erfahrung und fasst deshalb die notwendigen Tätigkeiten in der richtigen Reihenfolge zur PROC "Geschirrspülen" zusammen.

In einer Computersprache ist es ebenso. Eine Gruppe von Tätigkeiten, die zusammen eine größere Aufgabe erledigen sollen, werden in einer PROC oder FUNC kombiniert, der man dann einen Namen gibt. Per Aufruf des Namens wird diese dann ausgeführt (mit einigen Besonderheiten, die später erläutert werden). Allgemein wird das als FUNC- bzw. PROC-Aufruf bezeichnet. Zuvor allerdings muss die PROC oder FUNC bereits definiert worden sein, denn sonst kann sie ja nicht aufgerufen werden.

Wo liegt nun der Unterschied zwischen Prozeduren und Funktionen? Beide vollziehen eine Reihe an geordneten Schritten, um eine Aufgabe zu erfüllen. Warum also zwei ver-

schiedene Namen für die gleiche Sache? Weil sie nicht exakt gleich sind! FUNCs haben eine zusätzliche Fähigkeit; sie erfüllen ihre Aufgabe und geben einen Wert zurück.

In der Tabelle ist "Kontostand prüfen" als Beispiel für eine FUNC aufgeführt. Warum? Will man den Kontostand überprüfen, müssen einige Arbeitsschritte erledigt werden, um die einzelnen Buchungen abzurechnen. Das Ergebnis wird dann als Zahl von der FUNC ausgegeben und kann weiter verarbeitet werden.

Soll aus "Geschirr spülen" eine FUNC werden, muss die Anweisung auf "Muss das Geschirr gespült werden?" geändert werden. Dabei hofft man darauf, dass die gefragte Person die Frage mit 'Ja' beantwortet und - falls nötig - das Geschirr spült. Dadurch wird dann das Geschirr gespült (wie in der PROC), aber es wird auch ein Wert zurückgegeben (ob Spülen notwendig ist oder nicht). Und das macht es zu einer FUNC.

Anmerkung: Für den Rest des Handbuches wird das Wort "Routine" anstelle von Prozedur oder Funktion benutzt. Steht dann doch irgendwann "PROC" oder "FUNC", bezieht sich die dort dargelegte Idee speziell auf diese Art der Routine.

6.1 PROCeduren

Mit PROCs werden mehrere Anweisungen in einem Block zusammengefasst. Der Block wird dann mit Namen versehen und kann zur Erledigung der Aufgabe aufgerufen werden. Um PROCs in Action! benutzen zu können, muss man zwei Dinge wissen:

- PROC deklarieren
- PROC aufrufen

Die folgenden drei Punkte zeigen, wie das gemacht wird. Es folgen einige Beispiele zu PROCs in Action!.

6.1.1 PROC deklarieren

Das Schlüsselwort 'PROC' zeigt den Start einer PROC-Deklaration an. PROC-Konstruktionen sehen immer wie eine Gruppe von Anweisungen aus, die unter einem Namen zusammengefasst sind und am Anfang noch zusätzliche Informationen beinhalten; am Ende steht stets ein RETURN. Der formale Aufbau dieser Konstruktion:

```
PROC <Kennung>{=<Adr>}({<Parameterliste>})
  {<Variablendeklaration>}
  {<Anweisungenliste>}
RETURN
```

wobei

PROC	das Schlüsselwort ist, das die Deklaration einer Prozedur anzeigt.
<Kennung>	der Name der PROC ist.
<Adr>	die mögliche Startadresse der PROC[51]
<Parameterliste>	die Liste der von der PROC benötigten Parameter ist[52].

51) Kapitel IV, Abschnitt 9.3.
52) Kapitel IV, Abschnitt 6.4.

<Variablendeklaration> die Liste der Variablen ist, die lokal für diese PROC deklariert werden[53].
<Anweisungenliste> die Liste der Anweisungen in dieser PROC ist.
RETURN das Ende der PROC kennzeichnet[54].

Anmerkung: <Parameterliste>, <Variablendeklaration> und <Anweisungenliste> sind möglich, aber nicht unbedingt erforderlich. Man wird sicher einige davon verwenden, aber die folgende Deklaration einer PROC ist auch zulässig:

```
PROC Nix()     ; die Klammern müssen sein
RETURN
```

Diese "leere" PROC bewirkt natürlich nichts, kann aber nutzbringend eingesetzt werden, wenn man Programme entwickeln will, die aus vielen PROCs bestehen. Hat man im Programm z.B. den PROC-Aufruf "Verzinsung" verwendet, aber noch keine entsprechende Routine programmiert, so wird das Programm diese leere PROC abarbeiten. Den Rest des Programms kann man bereits auf Funktion testen, ohne dass der Fehler "Undeclared Variable" (nicht deklarierte Variable) auftritt.

Vorerst bleiben die <Parameterliste> und das 'RETURN' in der allgemeinen Form außerhalb der Betrachtung; das folgt später. Der Rest schaut eigentlich sehr bekannt aus, deshalb dazu folgendes Beispiel:

```
PROC Ratespiel()
; *** Die Prozedur spielt mit dem Anwender ein
; Ratespiel, das eine UNTIL-Schleife verwendet

  BYTE num,               ; die zu erratende Zahl
       guess              ; die geratene Zahl

  PrintE("Willkommen beim Ratespiel! Ich ")
  PrintE("denke mir eine Zahl zwischen 0 und 100.")
  num=Rand(101)               ; gibt die zu ratende Zahl
    DO                        ; Start der Schleife
      Print("Welche Zahl ? ")
      guess=InputB()          ; geratene Zahl
      IF guess<num THEN       ; Zahl zu klein
        PrintE("Zu klein, nochmal!")
      ELSEIF guess>num THEN   ; Zahl zu gross
        PrintE("Zu gross, nochmal!")
      ELSE                    ;richtige Zahl
        PrintE("Glueckwunsch!!!")
        PrintE("Du hast's geschafft!!!")
      FI                      ; Ende der Tests
      UNTIL guess=num         ; Schleifenkontrolle
    OD                        ; Ende der Schleife
RETURN                        ; Ende der PROC Ratespiel
```

53) Kapitel IV, Abschnitte 3.4.1 und 6.3.
54) Kapitel IV, Abschnitt 6.1.2.

Dies ist das bereits bekannte Beispiel aus Abschnitt 5.2.4.3. Und nun sieht man auch, warum PROC- und VAR-Deklarationen in diesem Programm verwendet werden. Ein Action!-Programm braucht die PROC- bzw. FUNC-Deklaration zum Kompilieren. Dieses Beispiel hat eine PROC-Deklaration und ist damit ein gültiges Action!-Programm, das kompiliert und benutzt werden kann. Seine Ausgabe sieht immer noch so aus:

```
Willkommen zum Ratespiel!
Ich denke mir eine Zahl zwischen 0 und 100.
Welche Zahl ? 50
Zu klein, nochmal!
Welche Zahl ? 60
Zu gross, nochmal!
Welche Zahl ? 55
Zu klein, nochmal!
Welche Zahl ? 57
Glueckwunsch!!!
Du hast's geschafft!!!
```

Sieht man sich nun das vorherige Beispiel an, erkennt man 'RETURN' als letzte Anweisung. Jetzt wird beleuchtet, warum es gerade dort steht.

6.1.2 RETURN

RETURN weist den Compiler an, die PROC zu verlassen und dorthin zurückzukehren, von wo die PROC aufgerufen wurde. Wurde die PROC von einem Programm aufgerufen, fährt es mit der Anweisung nach dem Aufruf fort. Kompiliert man nur eine PROC quasi als Programm, geht die Kontrolle anschließend an den Monitor zurück.

Warnung: Das Fehlen eines RETURN kann der Compiler nicht erkennen. Vergisst man ein RETURN, kann alles Mögliche mit dem Programm passieren. Das gilt auch für ein fehlendes RETURN am Ende einer FUNC.

Es kann sogar mehrere RETURNs in einer PROC geben. Verwendet man z.B. in einer PROC eine IF-Anweisung mit mehreren ELSEIF, dann kann man mit RETURN bei einem dieser ELSEIF bereits die PROC beenden. Das folgende Beispiel demonstriert das.

```
PROC testcommand()
; *** Diese PROC testet eine Eingabe auf Zulässigkeit.
; Erlaubt sind 0, 1, 2 und 3.
; Wird etwas anderes eingegeben, erfolgt eine Fehler-
; meldung und die Kontrolle wird dahin zurückgegeben,
; von wo die PROC aufgerufen wurde.

    BYTE cmd

    Print("Command>> ")
    cmd=InputB()
    IF cmd>3 THEN
      PrintE("Command Input ERROR")
      RETURN ;  Ruecksprung ohne zu testen
    ELSEIF cmd=0 THEN
      Print("<statement 0>")
```

```
    ELSEIF cmd=1 THEN
      Print("<statement 1>")
    ELSEIF cmd=2 THEN
      Print("<statement 2>")
    ELSEIF cmd=3 THEN
      Print("<statement 3>")
    FI
RETURN
```

Wichtig ist das RETURN nach der ersten Bedingung, die auf eine ungültige Eingabe hin überprüft. Schließlich sollen nicht erst alle Tests durchgegangen werden, bevor eine unzulässige Eingabe festgestellt wird. Deshalb wird die Fehlermeldung sofort ausgegeben und mit RETURN aus der PROC hinausgesprungen.

6.1.3 Prozeduren aufrufen

Bisher wurden einige PROC-Aufrufe ohne nähere Erläuterung präsentiert. Meist benutzt man eine Routine aus der Library per PROC-Aufruf. Die allgemeine Form ist einfach:

<Kennung> ({<Parameterliste>})

wobei

<Kennung>	der Name der PROC ist, die aufgerufen werden soll.
<Parameterliste>	die Werte enthält, die als Parameter an die PROC übergeben werden sollen.

Es folgen einige Beispiele. Die Parameter sind im Moment ohne Belang, ihnen ist ein Extraabschnitt gewidmet.

```
PrintE("Willkommen bei GoodByteXL!")
PrintE("Fan der ATARI 8-Bit-Computer.")
Fakultaeten()
guessuntil()

BYTE z
CARD add
signoff(add,z)
```

Natürlich müssen alle PROCs bereits deklariert worden sein, sonst könnten sie hier nicht verwendet werden. 'PrintE' ist eine Library-PROC, die in der Action!-Library enthalten ist und deshalb nicht deklariert werden muss. Nicht vergessen, dass PROCs immer runde Klammern brauchen, auch wenn keine Parameter übergeben werden. Soll eine aufgerufene PROC Parameter[55] verarbeiten, dürfen beim Aufruf nicht mehr Parameter übergeben werden, als in der Deklaration festgelegt wurde (aber weniger sind möglich).

55) Kapitel IV, Abschnitt 6.4.

6.2 FUNCtionen

Wie bereits im Überblick zu PROC und FUNC erwähnt, besteht der grundlegende Unterschied zwischen beiden darin, dass eine FUNC einen Wert zurückliefert. Deshalb sind Deklaration und Aufruf einer FUNC etwas anders als bei einer PROC. Da FUNCs einen numerischen Wert liefern, können sie nur dort benutzt werden, wo Zahlen zugelassen sind (z.B. in arithmetischen Ausdrücken).

6.2.1 Deklaration einer FUNCtion

Die Deklaration einer FUNC ist ähnlich der einer PROC, aber mit folgendem Unterschied. Es muss angegeben werden, welche Art von Zahl die FUNC liefert (BYTE, CARD oder INT) und was diese Zahl darstellt. Die allgemeine Form lautet:

<(Art)> FUNC<Kennung>{=<Adresse>} ({<Parameterliste>})
{<Variablendeklaration>}
{<Anweisungsliste>}
RETURN (<arithmetischer Ausdruck>)

wobei

<Art>	der Datentyp des Wertes ist, den die FUNC liefert.
FUNC	das Schlüsselwort ist, das die Deklaration einer FUNC anzeigt.
<Kennung>	der Name der FUNC ist.
<Adresse>	die optionale Startadresse der FUNC ist[56].
<Parameterliste>	die von der FUNC benötigten Parameter enthält[57].
<Variablendeklaration>	die Liste der lokalen Variablen für die FUNC ist[58].
<Anweisungsliste>	die Liste der Anweisungen in dieser FUNC ist.
RETURN	das Ende der FUNC anzeigt.
<arithmetischer Ausdruck>	der Wert ist, den die FUNC liefern soll.

Wie bei der Deklaration einer PROC sind <Parameterliste>, <Variablendeklaration> und <Anweisungsliste> optional. Bei PROCs war es nur in einem Fall sinnvoll, diese wegzulassen. Bei FUNCs gibt es dafür mehr Verwendungen, wie die folgenden Beispiele zeigen:

Beispiel 1

```
CARD FUNC square(CARD x)
       RETURN (x*x)
```

Diese FUNC nimmt eine CARD-Zahl und gibt das Quadrat davon zurück. Die Parameterliste bleibt im Moment außen vor und wird ein wenig später erläutert. Zuvor wurde schon dargelegt, dass der zurückgelieferte Wert in Form eines arithmetischen Ausdrucks ausgegeben wird. Im Beispiel 1 geschieht dies in "(x*x)".

Im folgenden Beispiel wurde dafür einfach nur ein VAR-Name benutzt.

56) Kapitel IV, Abschnitt 9.5.
57) Kapitel IV, Abschnitt 6.4.
58) Kapitel IV, Abschnitt 3.4.1 VAR-Deklaration und 6.3 Geltungsbereich.

Beispiel 2

```
BYTE FUNC getcommand()
; *** Diese FUNC liest eine eingegebene Zahl ein und gibt
; sie wieder aus, wenn sie den Wert 1 - 7 hat.
; Andernfalls wird der Anwender zur erneuten
; Eingabe aufgefordert.

  BYTE command, ; diese VAR enthaelt die Eingabe
       error    ; wird im Falle eins Fehlers auf 1 gesetzt
  DO
    Print("Eingabe> ")
    command=InputB()
    IF command<1 OR command>7 THEN ; ungueltige Eingabe
      error=1
      PrintE("Command Error: Nur 1-7 erlaubt.")
    ELSE                           ; gueltige Eingabe
      error=0
    FI
    UNTIL error=0 ; Schleife beenden, falls Eingabe gueltig
  OD
RETURN (command)
```

Anmerkung: Die runden Klammern um den <arithmetischen Ausdruck> in der RETURN-Anweisung sind unabdingbar.

Das war ein einfaches Beispiel. FUNCs können aber auch für viel kompliziertere Operationen eingesetzt werden. Doch selbst stark verschachtelte FUNCs müssen sich an dieser allgemeinen Form orientieren.

6.2.2 RETURN

Wie man an der allgemeinen Form der Deklaration einer FUNC sehen konnte, wird RETURN nicht in der gleichen Weise wie in der Deklaration einer PROC eingesetzt. In FUNCs wird RETURN ein <arithmetischer Ausdruck> nachgestellt. Diese Eigenschaft ermöglicht einer FUNC die Rückgabe eines Wertes. Versucht man einen <arithmetischen Ausdruck> nach dem RETURN einer PROC zu setzen, wird ein Fehler angezeigt, weil eine PROC keinen Wert zurückliefern kann.

Obwohl es Unterschiede zwischen den RETURNs einer FUNC und einer PROC gibt, existiert doch auch eine praktische Ähnlichkeit. Man kann sowohl in FUNCs als auch in PROCs mehr als ein RETURN einsetzen. Das folgende Beispiel zeigt die Anwendung von mehreren RETURNs in einer FUNC:

Vorgaben: Beispiel 1 im Abschnitt 6.2.1 gab das Quadrat einer CARD-Zahl aus, prüfte aber nicht auf Überlauf. Quadriert man 256, so erhält man den Wert 65536, der um 1 größer ist als der maximal zulässige Wert für eine CARD. Es gibt zwei Möglichkeiten, dieses Problem zu lösen:

1. Festlegen der zu quadrierenden Zahl als BYTE, was verhindert, dass eine Zahl größer als 255 eingegeben werden kann.

2. Den Überlauf in der FUNC selbst prüfen.

Das folgende Beispiel veranschaulicht die zweite Methode:

```
CARD FUNC square(CARD x)
; ***Diese FUNC testet 'x' auf Ueberlauf und gibt das
; Quadrat einer zulaessigen Eingabe aus. War die Eingabe
; nicht zulaessig, gibt die FUNC eine Fehlermeldung aus
; und liefert den Wert 0 zurueck.

  IF x>255 THEN    ;die Zahl verursacht einen Ueberlauf
    PrintE("Zahl zu gross")
    RETURN (0)     ;gibt null zurueck
  FI
RETURN (x*x)       ;liefert das Quadrat von 'x'
```

Ist das nicht einfach? Die Verwendung mehrerer RETURNs kann sehr nützlich sein, wenn verschiedene Bedingungen abgetestet werden, die unterschiedliche Werte zurückliefern sollen.
Anmerkung: Wie schon im Abschnitt 6.1.2 angemerkt, kann der Compiler nicht erkennen, wann ein RETURN fehlt. Deshalb muss man dafür sorgen, dass immer alle notwendigen RETURNs vorhanden sind.

6.2.3 Aufruf von FUNCtionen

Bisher wurden zwei Beispiele eines FUNC-Aufrufs präsentiert. Im Abschnitt 5.2.4.2 (WHILE) Beispiel 2 und im Abschnitt 5.2.4.3 (UNTIL) Beispiel 1. Sieht man sich diese Beispiele genauer an, wird man darin diese Zeilen finden:

```
num=Rand(101)
guess=InputB()
```

Das erste Beispiel für einen FUNC-Aufruf benötigt Parameter, im zweiten Beispiel findet man einen Aufruf ohne Parameter. Die FUNCs 'Rand' und 'InputB' stammen aus der Library. 'Rand' gibt eine Zufallszahl zwischen 0 und der angegebenen Zahl (hier 101) minus 1 aus. 'InputB' liest ein BYTE vom vordefinierten Gerät ein (hier Tastatur). Beide geben einen Wert zurück. Da dieser Wert ja irgendwo übergeben werden muss, müssen Aufrufe von FUNCs in einem arithmetischen Ausdruck stehen. In den zwei obigen Beispielen bestehen die arithmetischen Ausdrücke nur aus dem FUNC-Aufruf selbst und werden in einer zuordnenden Anweisung verwendet (zulässige Anwendung für arithmetische Ausdrücke). FUNC-Aufrufe können in jedem beliebigen arithmetischen Ausdruck verwendet werden, mit einer Ausnahme:

Ein FUNC-Aufruf darf nie in einem arithmetischen Ausdruck verwendet werden, wenn dieser Ausdruck als Parameter in einer Deklaration oder im Aufruf einer Routine benutzt wird.

Beispiel: x=square(2*Rand(50)) ; unzulässig

Hier sind einige Beispiele für zulässige FUNC-Aufrufe:

```
x=5*Rand(201)
c=square(x)-100/x
IF ptr<>Peek($8000)
chr=uppercase(chr)
```

'Peek' und 'Rand' sind Funktionen aus der Library und müssen daher nicht deklariert werden. Dagegen sind 'square' und 'uppercase' vom Programmierer geschriebene FUNCs und müssen vor dem ersten Aufruf deklariert werden.

Anmerkung: Obwohl es nicht gerade sinnvoll ist, kann man FUNCs wie PROCs aufrufen. Die zurückgegebenen Werte werden aber ignoriert.

6.3 Geltungsbereich von Variablen

In Action! kann eine VAR verschiedene Geltungsbereiche haben. Beim Programmieren wird festgelegt, in welchem Teil des Programms eine VAR benutzt werden darf und in welchem nicht. Das nachfolgende Programm demonstriert das am konkreten Beispiel:

```
MODULE     ; jetzt einige globale VARs deklarieren

CARD numgames=[0],   ; Anzahl der gespielten Spiele
     goal=[10],      ; Anzahl, die zu schlagen ist
     beatgoal=[0]    ; Anzahl, wie oft goal geschlagen wurde

PROC intro()
; *** Diese PROC gibt die Spielanleitung
; auf dem Bildschirm aus.

  CARD ctr
  PrintE("Willkommen zum Ratespiel. Ich")
  PrintE("denke mir eine Zahl")
  PrintE("von 0 bis 100.")
  PutE()
  PrintE("Du brauchst nur Deine Zahl nach")
  PrintE("der Aufforderung eingeben.")
  PutE()
  PrintE("Ich zaehle die Spiele mit")
  PrintE("und zeige an, wie oft Du in")
  PrintE("frueheren Versuchen gebraucht hast,")
  PrintE("die Bestmarke zu schlagen.")
  PutE()
  PrintE("Aber erst mal gib mir")
  PrintE("Deine Bestmarke ein.")
  PutE()
  Print(" Am besten hier --> ")
  goal=InputC()
  FOR ctr=0 to 2500 ; Warteschleife, die dem Spieler
    DO              ; das Gefuehl eines Ablaufs geben soll.
    OD
  Put($7D)          ; Bildschirm loeschen
RETURN
```

```
PROC tally()
; *** Diese PROC gibt den aktuellen Spielstand aus

  Print("Du hast ")
  PrintC(numgames)
  PrintE(" Spiele gespielt.")
  PutE()
  Print("Davon hast Du in ")
  PrintC(beatgoal)
  PrintE(" Spielen")
  PrintE("Deine Bestmarke")
  Print("von ")
  PrintC(goal)
  PrintE(" geschlagen.")
  PutE()
RETURN      ; Ende der PROC tally

PROC playgame()

  CARD numguesses,     ; Anzahl der Versuche
       ctr             ; Zähler der Warteschleife

  BYTE num,            ; zu ratende Zahl
       guess           ; geratene Zahl
  PrintE("Speichere Deine Bestmarke...")
  FOR ctr=0 TO 4500   ; Verzoegerung, die den Spieler glauben
    DO                ; laesst, der Computer naehme die Zahl
    OD
  PutE()
  PrintE(" Auf geht's!")
  PutE()
  num=Rand(101)         ; waehlt die zu ratende Zahl
  numguesses=0          ; Versuche auf 0 setzen
  DO                    ; Start der UNTIL-Schleife
    Print("Welche Zahl rätst Du ? ")
    guess=InputB()        ; geratene Zahl uebernehmen
    numguesses==+1        ; Anzahl Versuche um 1 erhoehen
    IF guess<num THEN     ; Versuch zu tief
      PrintE("Zu tief, versuch's noch mal!")
    ELSEIF guess>num THEN  ; zu hoch
      PrintE("Zu hoch, versuch's noch mal!")
    ELSE                  ; richtige Zahl
      PrintE("Glueckwunsch!!!")
      Print("Du hast's geschafft in ")
      PrintCE(numguesses)
      PrintE(" Versuchen!")
      IF numguesses<goal THEN
         beatgoal==+1
      FI
    FI ; Ende des Ratetests
    UNTIL guess=num
```

```
  OD     ; Ende der UNTIL-Schleife
RETURN   ; Ende der PROC playgame

BYTE FUNC stop()
; *** Diese Funktion fragt ab, ob der Spieler noch mal
; spielen will

  BYTE again

  PrintE("Noch mal")
  Print("spielen ? (J oder N) ")
  again=GetD(1) ; Antwort des Spielers von Tastatur ueber-
                ; nehmen hiermit wird ein Return als erste
                ; Eingabe für das naechste Spiel verhindert
  PutE()
  IF again='N or again ='n THEN  ; will nicht
     RETURN (1)                  ; spielen
  FI
  Put($7D)                       ; Bildschirm loeschen
RETURN (0)                       ; Ende der FUNC stop
PROC main()

  Close(1)           ; zur Sicherheit
  Open(1,"K:",4,0)   ; von Tastatur lesen
  intro()            ; Anleitung ausgeben
  DO
  numgames==+1       ; Anzahl der Spiele hochzaehlen
  playgame()         ; das Spiel einmal spielen
  tally()            ; zeigt gesamten Spielstand
  UNTIL stop()       ; nicht mehr spielen
  OD
  PutE()
  PrintE("Spiel bald mal wieder!")
  Close(1)           ; K: schliessen
RETURN               ; Ende der PROC main (Hauptprogramm)
```

Die Tabelle zeigt, wie das Programm Variablen benutzt. Es werden Namen, Geltungs-bereich, Verfügbarkeit und Benutzung der Variablen in jeder Routine angegeben.

Abkürzungen: V = Verfügbarkeit in Routine, B = Benutzung in der Routine

Variable		**PROC**	**PROC**	**PROC**	**FUNC**	**FUNC**
Name	**Geltung**	**playgame**	**intro**	**tally**	**stop**	**main**
numgames	global	V	V	V B	V	V B
goal	global	V B	V B	V B	V	V
beatgoal	global	V B	V	V B	V	V
numguesses	lokal	V B				

Variable		**PROC**	**PROC**	**PROC**	**FUNC**	**FUNC**
Name	**Geltung**	**playgame**	**intro**	**tally**	**stop**	**main**
num	lokal	V B				
guess	lokal	V B				
ctr	lokal	V B				
again	lokal				V B	
ctr	lokal		V B			

Globale Variablen können für jede Routine zugelassen werden, lokale Variablen dagegen nur für die Routinen, in denen sie deklariert werden. So gibt es zwei lokale Variablen mit Namen 'ctr'. Eine in der 'PROC playgame' und die andere in der 'PROC intro'. Obwohl sie denselben Namen tragen, handelt es sich nicht um die gleiche Variable. Die zwei 'ctr' haben unterschiedliche lokale Geltungsbereiche, da sie ja in verschiedenen Routinen deklariert wurden.

6.4 Parameter

Mit Parametern ist es möglich, Werte an eine Routine zu übergeben. Wozu das erforderlich ist? Man könnte ja dafür und für den Austausch von Werten zwischen Routinen globale VARs einsetzen. Nun, es gibt zwei Gründe dafür:

- Die vielseitigere Nutzung von Routinen.
- Die Manipulation von VAR-Werten innerhalb der Routine, ohne den Wert einer globalen VAR ändern zu müssen.

Beide Möglichkeiten werden in der obigen Reihenfolge ausführlich erläutert. Aber zuvor noch ein Blick auf die Form einer Parameterliste.

Parameter in PROC- oder FUNC-Deklarationen:

({<Variablen Dekl.>}l:,<Variablen Dekl.>:l)

wobei

<Variablen Dekl.> eine normale Variablendeklaration ist; mit Ausnahme der Option '=<Adresse> oder [<Konstante>]'.

Beispiele:

```
PROC test(BYTE chr,num,i, CARD x,y)
  INT FUNC docommand(INT cmd, CARD ptr, BYTE offset)
  CARD FUNC square(BYTE x)
PROC jump()
```

Parameter in PROC- oder FUNC-Aufrufen:

({<Arithm. Ausdr.>}l:,<Arithm. Ausdr.>:l)

wobei

<Arithm. Ausdr.> ein arithmetischer Ausdruck ist.

Beispiele:

```
test(cat,dog,ctr,2500,$8d00)
sqr=square(num)
jump()
x=docommand(temp,var,'A)
```

Anmerkung: Eine Routine kann bis zu 8 Parameter übernehmen. Werden mehr verwendet, führt das zu einem Compilerfehler.

Das folgende Beispiel zeigt auf, wie Parameter verwendet werden und verdeutlicht die Vielseitigkeit in der Nutzung von Routinen.

Die folgende FUNC prüft, ob die BYTE-Variable 'chr' ein Kleinbuchstabe ist. Wenn ja, gibt die FUNC den Großbuchstaben dazu aus. Andernfalls wird von der FUNC einfach 'chr' ausgegeben. Beachtenswert: 'chr' wird nirgendwo deklariert. Im Anschluss an das Beispiel wird erklärt, wo die VAR hätte deklariert werden müssen.

```
BYTE FUNC lowertoupper()

  IF chr>='a AND chr<='z THEN  ; $20 ist der Offset
    RETURN (chr-$20)           ; von Klein- zu Grossbuch-
  FI                           ; staben im ATASCII-Code
RETURN (chr)
```

Nun gilt es noch zu bestimmen, wo 'chr' deklariert werden muss. Klar ist, dass es sowohl global als auch nur lokal für 'lowertoupper' deklariert werden kann. Wenn es nur lokal deklariert wird, wie kann dann ein Wert übergeben werden? Es scheint keinen Weg zu geben, 'chr' als lokale VAR zu deklarieren, da dann die FUNC selbst der VAR einen Wert übergeben müsste. Und das ist nicht gerade das, was die FUNC tun soll. 'lowertoupper' soll so ähnlich wie

chr=lowertoupper()

aufgerufen werden, dann 'chr' testen und ggf. in einen Großbuchstaben verwandeln. Also darf 'chr' nicht lokal deklariert werden. Wie wäre es mit einer globalen Deklaration? Das würde wie gedacht funktionieren. Denn nun ist 'chr' beim Aufruf der FUNC und 'chr' in der FUNC selbst die gleiche globale VAR. Es gibt bei der Deklaration von 'chr' als globaler VAR nur einen Nachteil: Bei jeder Verwendung von 'lowertoupper' bekommt man den Großbuchstaben von 'chr'. Wenn die VAR 'cat' in einen Großbuchstaben umgewandelt werden soll, müsste man das etwa so angehen:

chr=cat
chr=lowertoupper()
cat=chr

Eine ziemlich umständliche Methode, wenn man viele verschiedene VARs in Großbuchstaben umwandeln will. Und, falls 'lowertoupper' in ein anderes Programm eingebaut werden sollte, müsste man gleichfalls eine globale VAR namens 'chr' deklarieren.

Wie wäre es, wenn man 'chr' als Parameter festlegte und an die FUNC übergäbe? Wie das?! Na, einfach so:

```
BYTE FUNC lowertoupper(BYTE chr)  ; <- Deklaration als Parameter
  IF chr>='a AND chr<='z THEN
     RETURN (chr-$20)
  FI
RETURN (chr)
```

Aber wie soll das aufgerufen werden? Das ist leicht. Man übergibt die zu prüfende VAR einfach als Parameter. Beispiele:

```
chr=lowertoupper(chr)
cat=lowertoupper(cat)
var=lowertoupper('a)
```

Durch Umwandlung von 'chr' in einen Parameter für die FUNC wird es möglich, jede beliebige Variable in jedem Programm zu prüfen. Das ist möglich, weil 'lowertoupper' nun für sich allein steht, also unabhängig ist. Es benutzt keine irgendwo deklarierten VARs, und man kann sogar VARs an 'lowertoupper' zum Testen übergeben. Damit sind alle Probleme bei der Deklaration von 'chr' als lokale oder globale VAR überwunden. Und genau das versteht man unter "vielseitiger Nutzung von Routinen".

Der zweite Grund für die Parameter ist etwas schwieriger zu erläutern. Es soll so klar wie möglich an einem Beispiel dargelegt werden. Die folgende PROC übernimmt zwei CARD-Zahlen, teilt die erste Zahl durch die zweite und gibt das Ergebnis aus:

```
PROC division(CARD num,div)

  num==/div          ; ersetzt num durch num/div
  PrintC(num)        ; gibt num aus
RETURN
```

Und nun wird die PROC 'division' in einem Programm verwendet:

```
PROC main()

  CARD ctr,
       number= [713]

  FOR ctr=1 to 10
    DO
      PrintC(number)
      Print("/")
      PrintC(ctr)
      Print(" = ")
      division(number,ctr)
      PutE()
    OD
RETURN
```

Ausgabe:

```
713/1 = 713
713/2 = 356
713/3 = 237
713/4 = 178
```

```
713/5 = 142
713/6 = 118
713/7 = 101
713/8 = 89
713/9 = 79
713/10 = 71
```

Beachtenswert: 'number' bleibt konstant, obwohl 'num' sich verändert. Der Wert von 'number' wird an 'num' übergeben, sobald die PROC aufgerufen wird. Aber der Wert von 'num' wird nicht nach 'number' zurückgeschrieben, wenn die PROC beendet wird. Würde der Wert von 'num' an 'number' zurück übergeben, sähe die Ausgabe so aus:

```
713/1 = 713
713/2 = 356
356/3 = 118
118/4 = 29
29/5 = 5
5/6 = 0
0/7 = 0
0/8 = 0
0/9 = 0
0/10 = 0
```

Der Datenfluss per Parameter ist nur in einer Richtung möglich. Daten können an eine Routine per Parameter übergeben werden, aber es ist grundsätzlich nicht möglich, Daten mit Hilfe von Parametern aus der Routine auszugeben. Soll ein Wert von einer Routine zurückgegeben werden, muss das als FUNC konzipiert werden. Dann kann ein Wert mit der RETURN-Anweisung der FUNC zurückgeliefert werden. Sollen sogar mehrere Werte zurückgegeben werden, muss man globale VARs verwenden oder Zeiger (pointer) als Parameter übergeben[59].

Anmerkung zur Zuordnung der Parameter:
Beim Aufruf einer Routine, die Parameter hat, wird der erste übergebene Parameter der ersten VAR in der VAR-Liste der Routinendeklaration zugeordnet, der zweite Parameter der zweiten VAR usw. Man kann weniger Parameter als von der Routine gefordert übergeben, aber nicht mehr. Sind also 5 Parameter in der Deklaration bestimmt, können 0 bis 5 Parameter angegeben werden. Dadurch lassen sich Routinen entwickeln, die eine unterschiedliche Anzahl an Parametern benutzen, abhängig von der Aufgabenstellung. Tipp: Wenn so etwas programmiert wird, sollte der erste Parameter die Anzahl der zu übergebenden Parameter enthalten.

Anmerkung zur Kompatibilität der VAR-Typen:
Sollten der per Parameter übergebene Wert und der von der Routine erwartete Wert unterschiedlichen Datentyps sein, führt das nicht zu einem Compilerfehler! Action! verfügt nämlich bei Parametern über eine Kompatibilität hinsichtlich der Datentypen. Übergibt man z.B. eine CARD, wenn die PROC eine BYTE verlangt, wird das LSB der CARD in der BYTE abgelegt. Die PROC arbeitet dann so weiter, als ob man eine BYTE übergeben hätte[60].

59) Kapitel IV, Abschnitt 9.5.
60) Kapitel IV, Abschnitt 4.

Anmerkung zu den VAR-Typen bei Parametern:
Die nachfolgenden Typen sind bei Parametern zulässig:

- VARs vom elementaren Datentyp
- Verweise auf Arrays, Pointer und Records.
- Namen von Arrays, Pointern und Records.

Im Falle des dritten Anstrichs werden die verwendeten Namen als Zeiger (Pointer) auf das erste Element, den Wert oder auf das erste Feld in der benannten VAR gesetzt.

7 Compiler-Direktiven

Compiler-Direktiven unterscheiden sich von den Standardkommandos dadurch, dass sie nur beim Kompilieren und nicht beim eigentlichen Programmablauf ausgeführt werden. Ein Kommando der Sprache, z.B. eine zuordnende Anweisung[61], wird ausgeführt, sobald der Monitor die Anweisung 'RUN' erhält; also wenn das Programm die Kontrolle über das Geschehen hat. Eine Compiler-Direktive wird ausgeführt, wenn im Monitor das Kommando 'Compile' eingegeben wird, also der Compiler und nicht das Programm die Kontrolle hat. Die Unterschiede werden jetzt herausgearbeitet.

7.1 DEFINE

Die Direktive DEFINE ist dem Ersetzen-Kommando (<Ctrl><Shift>S) des Editors sehr ähnlich. Allerdings wird ausschließlich während des Kompilierens ersetzt. Um darüber Klarheit zu gewinnen, hier die allgemeine Form:

DEFINE <Kenn>=<String-Konst.>{,<Kenn>=<String-Konst.>}

wobei

<Kenn> eine zulässige Kennung ist
<String-Konst.> eine in Action! zulässige String-Konstante ist (→Anführungszeichen).

DEFINE ist eigentlich zur Erzeugung des Objektcodes beim Kompilieren des Programms gar nicht erforderlich. Aber es ist sehr nützlich für die Übersichtlichkeit des Programms im Action!-Quelltext. Der Compiler ersetzt die <Kennung> jedes Mal dann durch <String-Konstante>, wenn die Kennung im Programmtext vorkommt. Kompiliert man z.B. ein Programm mit der Zeile

```
DEFINE size = "256"
```

so ersetzt der Compiler jedes 'size' durch '256'. Das ermöglicht einige interessante Optionen (und Probleme, falls verkehrt angewandt). Da DEFINE jeden beliebigen String ersetzen kann, können sogar die Schlüsselwörter selbst geändert werden! Gefällt einem z.B. das Schlüsselwort CARD nicht, dann kann man es beispielsweise durch FROG ersetzen. Dazu dient folgendes Kommando:

```
DEFINE CARD = "FROG"
```

Beim Kompilieren des Programmtextes erkennt der Compiler, dass er CARD durch FROG ersetzen soll und erledigt das dann auch.

---

61) Kapitel IV, Abschnitt 5.1.1.

Hier noch einige weitere Beispiele zum Format dieser Direktive:

```
DEFINE liston = "SET $49A=1"
DEFINE begin = "DO", end = "OD"
DEFINE one = "1"
```

Anmerkung: Auf keinen Fall die Anführungszeichen vor und hinter der String-Konstanten[62] vergessen.

Zur Verdeutlichung der Möglichkeiten von DEFINE eine Tabelle.

Anweisung	Erläuterung
DEFINE four = " 4 "	die Direktive
PrintBE(four)	gibt ' 4 ' und ein EOL aus
; four score	ersetzt 'four' durch ' 4 '
; four-score	'four-score' wird nicht geändert
PrintE("four score")	innerhalb von Anführungszeichen wird nie ersetzt

7.2 INCLUDE

Die Direktive INCLUDE ermöglicht das Einbinden von anderen Programmtexten in den zu kompilierenden Programmtext. Soll z.B. das Programm 'EAZEUGS.ACT', das Ein/Ausgabe-Funktionen erledigt, in andere Programme übernommen werden, muss nur das folgende Kommando in den Text einfügt werden:

```
INCLUDE "D1:EAZEUGS .ACT"
```

Anmerkung: Die Datei-Kennung muss in Anführungszeichen stehen!

Die gezeigte Anweisung muss natürlich erfolgen, bevor die E/A-Routinen aus der Datei 'EAZEUGS .ACT' verwendet werden. Das in diesem Beispiel angesprochene Laufwerk #1 muss natürlich vorhanden sein. Ohne Laufwerksangabe greift der Compiler auf Laufwerk #1 zu[63].

Mit INCLUDE lassen sich Dateien von jedem Speichergerät einlesen. Beispiele dazu:

```
INCLUDE "D:IOLIB.ACT"
INCLUDE "PROG1.DAT"
INCLUDE "C:"
```

Anmerkung: Die meisten DOS-Versionen verlangen die Datei-Kennung in Großbuchstaben.

Eine besonders nützliche Option der Direktive INCLUDE ist die Möglichkeit, sie auch in einem Programm verwenden können, das gerade beim Kompilieren per INCLUDE übernommen wird. Es sind also Schachtelungen möglich. Action! lässt dafür eine 6-fache Schachtelung zu. Allerdings unterliegen die Peripheriegeräte sowie die DOS-Versionen oft anderen, meist engeren Einschränkungen. Werden die Grenzen des DOS ignoriert,

62) Kapitel IV, Abschnitt 3.2.
63) Bei SpartaDOS und kompatible auf aktuellen Pfad.

führt das zu einem Fehler #161 (zu viele offene Dateien). Bei Kassettenlaufwerken ist mit INCLUDE nur eine offene Datei möglich, bei Diskettenstationen meist 3 (hängt vom DOS ab). Ist kein Programmtext im Editors gespeichert, wird also direkt von einem Speichermedium kompiliert, reduziert sich die Anzahl der Verschachtelungsebenen von INCLUDEs um eins.

Anmerkung: Leistungsfähige DOS erlauben mehr offene Dateien. SpartaDOS X[64], das mit einer Standardkonfiguration von 5 gleichzeitig geöffneten Dateien arbeitet, bietet dem Nutzer die Option auf 2 bis 16 offene Dateien.

7.3 SET

Die Direktive SET wird zur Veränderung von Speicherinhalten gebraucht. SET 'poket' beim Kompilieren einen neuen Wert in die angegebene Speicherstelle. Meist wird dieses Kommando zur Veränderung von Editor- oder Compiler-Optionen durch ein Anwenderprogramm benutzt. Aber man kann damit sogar User-, Betriebssystem- und Hardware-VARs verändern. Das Format sieht so aus:

SET <Adresse> = <Wert>

Anmerkung: <Adresse> und <Wert> müssen Compiler-Konstante sein!

Die Direktive SET hat die Aufgabe, die Speicherstelle <Adresse> auf den angegebenen <Wert> zu setzen. Ist der <Wert>, größer als 255, dann wird er in die Speicherstellen <Adresse> und <Adresse+1> geschrieben. Das liegt daran, dass 255 ($FF) die größte Zahl ist, die in einem BYTE gespeichert werden kann. Deshalb wird für eine größere Zahl eine zweite Speicherstelle hinzugenommen und der Wert im für den ATARI typischen LSB-MSB-Format abgelegt. Beispiele dazu:

```
SET $600=64     ; setzt Adresse $600 auf 64
SET max=16      ; setzt max auf 16
SET 10000=$FFFF ; setzt 10000 und 10001 auf $FFFF
SET $CF00=cat   ;setzt $CF00 und $CF01 auf @ cat

DEFINE add="$7000"
SET add=$42
```

Das letzte Beispiel zeigt eine DEFINEte numerische Konstante, die in einer SET-Anweisung verwendet wird. Da DEFINEs ja beim Kompilieren Konstante sind, sind sie für die Direktive SET zugelassen. Man muss nur die numerische Konstante DEFINEn, bevor man sie in einer SET-Anweisung benutzt.

Anmerkung: Bitte die Wirkung von SET beim Kompilieren nicht verwechseln mit der ziemlich ähnlichen Wirkung von Poke und PokeC beim Programmablauf.

7.4 MODULE

MODULE ist eine einfache Direktive. Ihre Form lautet:

MODULE

64) SpartaDOS X V. 4.4x, SPARTA.SYS Treiber.

Sie teilt lediglich dem Compiler mit, dass einige globale Variablen deklariert werden sollen. Die Direktive ist sehr nützlich, wenn lange Programme in viele Unterprogramme unterteilt sind, von denen jedes eigene globale VARs hat. Setzt man MODULE am Anfang jedes Unterprogramms ein, so fügt der Compiler alle globalen VARs der VAR-Tabelle hinzu.

Ein Programm muss nicht unbedingt die Direktive MODULE enthalten, weil der Compiler am Anfang immer ein MODULE annimmt, egal ob es gesetzt wurde oder nicht.

Die Deklaration von globalen VARs muss unmittelbar nach MODULE stehen oder gleich am Anfang des Programms (wo sie unmittelbar nach der vom Compiler angenommenen Direktive MODULE steht).

8 Erweiterte Datentypen

Die erweiterten Datentypen sorgen für die größere Flexibilität von Action! im Vergleich zu anderen Sprachen. Es wurde ja schon aufgezeigt, dass strukturierte Anweisungen Gruppierungen von einfachen Anweisungen manipulieren können und dabei die Möglichkeiten der Sprache an sich erweitern. So ähnlich manipulieren auch die zusätzlichen Datentypen Gruppierungen von elementaren Datentypen und vergrößern damit die Fähigkeiten der Sprache noch weiter.

Es gibt in Action! drei erweiterte Datentypen:

- Pointer (Zeiger)
- Arrays (Felder)
- Records (Sätze)

Die Typen werden nachfolgend in dieser Reihenfolge besprochen.

8.1 POINTER

Pointer (Zeiger) hört sich irgendwie nach Zeigestock an. Und genau das ist es auch. In Action! haben "Pointer" eine ganz ähnliche Bedeutung.

Pointer enthalten eine Speicheradresse und zeigen daher auf eine Speicherstelle. Man kann den Wert eines Pointers verändern und dadurch auf eine neue Speicherstelle zeigen. Der Pointer kann auf BYTE-, CARD- oder INT-Werte zeigen.

Irgendwie muss man den Compiler nun wissen lassen, auf welche Art von Wert ein gesetzter Pointer zeigen soll. Das wird mit der Deklaration des Pointers angegeben.

Zunächst werden die Methoden der Deklaration dargelegt und dann die Anwendung derselben erläutert. Ein Beispielprogramm demonstriert danach die Möglichkeiten.

8.1.1 Pointer-Deklaration

Die Form für die Deklaration eines Pointers ist der Form für die Deklaration einer Variablen des elementaren Datentyps ähnlich. Der Unterschied ist, dass dem Compiler mitgeteilt wird, dass die Variable ein Pointer und kein elementarer Datentyp ist.

<Typ> POINTER <Kennung>{=<Adresse>}l:,<Kennung>{=<Adresse}:l

wobei

<Typ> der elementare Datentyp der Information ist, auf den der Pointer zeigt.
POINTER das Schlüsselwort ist, das anzeigt, dass die deklarierten Variablen Pointer sind.
<Kennung> der Name der Pointer-Variablen ist.
<Adresse> angibt, auf welche Speicherstelle der Pointer zu Beginn zeigen soll. Der Pointer muss eine Compiler-Konstante sein.

Da eine Pointer-Variable stets eine Adresse beinhaltet, muss ein Wert von 0 - 65535 ($0 bis $FFFF) darin ablegbar sein. Deshalb werden Pointer als Zwei-Byte-Zahlen ohne Vorzeichen im LSB-MSB-Format abgespeichert. Das bedeutet, dass Pointer wie CARDs abgespeichert, aber als Adressen betrachtet werden können.

Zunächst einige Deklarationen:

```
BYTE POINTER ptr        ;deklariert ptr als Pointer auf
                        ;einen BYTE-Wert
  CARD POINTER cpl      ;deklariert cpl als Pointer auf
                        ;eine CARD
  INT POINTER ip=$8000  ;deklariert ip als Pointer auf
                        ;eine INT und setzt ihn auf die
                        ;Speicherstelle $8000.
```

8.1.2 Pointer-Manipulation

Pointer lassen sich in Action! für die Manipulation von vielen Dingen einsetzen. Das liegt einfach daran, dass Pointer leicht dazu verwendet werden können, auf verschieden Speicherstellen zu zeigen. Daher ist das Katalogisieren und Tabellarisieren von Informationen sehr einfach.

Das folgende Programm ist nur ein simples Beispiel, das eine Vorstellung davon vermitteln soll, was ein Pointer tatsächlich macht. Es demonstriert den Adress-Operator '^' in Verbindung mit Pointern. Nach dem Beispiel wird das '^' ausführlicher betrachtet.

```
PROC pointerusage()

  BYTE num=$E0,          ;deklariert und setzt
       chr=$E1           ;zwei BYTE-Variable
  BYTE POINTER bptr      ;deklariert einen Pointer
                         ;auf einen BYTE-Typ
  bptr= @ num            ;bptr zeigt jetzt auf num
  Print("bptr zeigt jetzt auf Adresse ")
  PrintF("%H",bptr)      ;gibt Adresse von num aus
  PutE()
  bptr^=255              ;schreibt 255 in die Speicherstelle,
                         ;auf die bptr zeigt (z.B. in num)
  Print("num = ")
  PrintBE(num)           ;zeigt, dass in num jetzt 255 steht
  bptr^=0                ;schreibt in num 0 ein
  Print("num = ")
  PrintBE(num)           ;num enthaelt jetzt 0
  bptr= @ chr            ;bptr zeigt jetzt auf chr
  Print("bptr zeigt jetzt auf Adresse ")
  PrintF("%H",bptr)      ;gibt chr's Adresse aus; man sieht,
```

```
                             ;dass bptr tatsaechlich geaendert ist
  PutE() ;
  bptr^='q                   ;schreibt 'q in die Speicherstelle,
                             ; auf die bptr zeigt (z.B. in chr)
  Print("chr = ")
  Put(chr)                   ;zeigt, dass chr tatsaechlich 'q ist
  PutE()
  bptr^='z                   ;aendert chr in 'z
  Print("chr = ")
  Put(chr) ;zeigt -> chr='z
  PutE()
RETURN
```

Bildschirmausgabe

```
bptr zeigt jetzt auf Adresse $00E0
num = 255
num = 0
bptr zeigt jetzt auf Adresse $00E1
chr = q
chr = z
```

Wie ersichtlich wird der Operator '^' benutzt, wenn ein Wert in die Speicherstelle geschrieben werden soll, auf die der Pointer zeigt. Die Zeile "bptr^=0" im Beispiel entspricht dem Ausdruck "num=0", weil 'bptr' zu der Zeit auf 'num' zeigt. Pointer-Angaben können auch in arithmetischen Ausdrücken verwendet werden:

x=ptr^

Merke, dass "PrintF(%H, bptr)" zulässig ist. Was also bedeutet, dass 'bptr' sowohl als CARD-Zahl als auch als Adresse verwendet werden kann. Das ist sehr nützlich beim Entfehlern von Programmen, da man so leicht herausfinden kann, worauf der Pointer tatsächlich zeigt.

8.2 ARRAYs (Felder)

Mit Hilfe von ARRAYs lässt sich eine Variablenliste sehr einfach manipulieren, indem man über den ARRAY-Namen in Verbindung mit einem Index auf die Liste zugreift. Die Variablen in der Liste müssen allerdings vom gleichen Datentyp sein, und es sind nur die elementaren Datentypen erlaubt. Der ARRAY-Name gibt an, welches ARRAY angesprochen werden soll; der Index gibt an, auf welches Element aus der Liste zugegriffen werden soll. Der Index besteht nur aus einer Zahl. Man kann als Angabe zu einem Element aus dem ARRAY sagen: "Ich brauche das n-te Element des ARRAYs x." Dabei ist 'n' der Index und 'x' der Name des ARRAYs. Im Folgenden wird die interne Darstellung eines ARRAYs erläutert. Danach wird gezeigt, wie ARRAYs deklariert und manipuliert werden und wo die Grenzen von ARRAYs unter Action! Liegen.

8.2.1 Deklaration eines ARRAYs

Es ist relativ einfach, ARRAYs in Action! zu deklarieren. Was aber noch lange nicht heißt, dass man eine vollständige Kontrolle über das hat, was dabei abläuft. Es gibt eine Menge Möglichkeiten für die Festlegung der unterschiedlichen Charakteristika eines ARRAYs einschließlich ihrer Adresse, Größe und sogar ihrer ersten Inhalte. Aufgrund dieser vielfältigen Optionen sieht die allgemeine Form etwas überladen aus. Die Beispiele werden aber die Verwirrung beseitigen.

<type> ARRAY <var init> I:,<var init>:I

wobei

<type>	der elementare Datentyp der Elemente des Arrays ist.
ARRAY	das Schlüsselwort für ein Array ist.
<var init>	die notwendige Information ist, um eine Variable als Array aus Elementen des Daten<type>s zu deklarieren.

<var init> hat dabei das Format:

<ident>{(size)}{=<addr> I [<values>] I <str const>}

wobei

<ident>	der Name der Variablen ist.
<size>	die Größe des Arrays ist und eine Konstante sein muss.
<addr>	die Adresse des ersten Elementes im Array ist und eine Compiler-Konstante sein muss.
[<values>]	die Anfangswerte der Elemente des Arrays setzt. Jeder Wert muss eine numerische Konstante sein.
<str const>	die Anfangswerte der Elemente des Arrays durch eine String-Konstante setzt; das erste Element gibt dann die Länge des Strings an.

Wie gesagt, es ist schon verwirrend! Aber nun werden die Schleier mithilfe einiger (hoffentlich) klarer Beispiele gelüftet:

```
BYTE ARRAY a,b          ;deklariert zwei Arrays mit BYTE Elemen-
                        ;ten, ohne dabei die Groesse festzulegen
INT ARRAY x(10)            ;deklariert 'x' als ein INT-Array
                           ;und legt die Groesse fest.

BYTE ARRAY str="Dies ist eine String-Konstante!"
                           ;deklariert 'str' als BYTE-Array und
                           ;fuellt es mit einem String

CARD ARRAY junk=$8000      ;deklariert 'junk' als CARD-Array,
                           ;das im Speicher bei $8000 beginnt,
                           ;ohne die Groesse zu bestimmen.

BYTE ARRAY tests= [4 7 18]      ; deklariert 'tests' als
                                ; BYTE-Array und setzt Werte
                                ; hinein.
```

Anmerkungen zum Programmieren: Soll ein ARRAY dimensioniert werden, nur Dezimalwerte verwenden, da der Compiler mit Hex-Werten fehlerhaften Code erzeugt.

Wo immer möglich, sollte die Größe eines Arrays dimensioniert werden. Aber es gibt auch einige Fälle, in denen man das nicht muss oder kann:

- Es ist nicht bekannt, wie groß das Array beim Programmablauf werden kann.

- Das Array wird bei der Deklaration mit einem String (durch '[<values>]' oder '<str const>') gefüllt und später nicht mehr erweitert.

Also daran denken, dass das erste Byte einer String-Konstanten die Länge des Strings enthält. Soll ein String verlängert werden, muss als Erstes das erste Byte auf die neue Länge geändert werden (welches immer das 0-te Byte des Arrays ist, das den String enthält).

8.2.2 Interne Darstellung

Die interne Darstellung eines Arrays ähnelt sehr stark der eines Pointers. Das liegt daran, das der Array-Name tatsächlich ein Pointer auf das erste Element des Arrays ist. Das Array selbst ist nur eine zusammengehörige Anzahl an Speicherstellen, die jede ein Element des Arrays enthalten. Die Größe einer Speicherstelle ist bestimmt durch den Datentyp der Elemente: 1-Byte-Stellen für BYTEs, 2-Byte-Stellen für CARDs und INTs. Auf jeden Fall eröffnen sich dadurch, dass der Name des Arrays ein Pointer ist, einige äußerst interessante Tricks. Welche, das wird in den Beispielen 2 bis 4 des nächsten Teils gezeigt.

8.2.3 Manipulation eines Arrays

Anwendung und Manipulation von Arrays ist nicht weiter schwer, wenn man weiß, wie Arrays deklariert werden und wie man auf die Elemente zugreift. Das Deklarieren wurde schon gezeigt. Nun also der Zugriff:

Beispiel 1

```
PROC reftest()

  BYTE x

  BYTE ARRAY nums(10)

  FOR x=0 TO 9                ;Da nums 10 Elemente hat, wird
                              ;von 0 bis 9 gezaehlt.
    DO
      nums(x)=x+'A            ;x-tes Element von nums wird auf
                              ;den Wert x+'A gesetzt.
      Put(nums(x))            ;Gibt das x-te Element von nums
                              ;als Zeichen aus.
      Print(" ")              ;Leerzeichen zwischen den Zeichen.
    OD
  PutE()
RETURN
```

Ausgabe von Beispiel 1: `A B C D E F G H I J`

Im obigen Programm wird zweimal auf das Array zugegriffen: Auf 'nums(x)' mit der zuordnenden Anweisung und auf 'nums(x)' als Parameter für die Library-PROC 'Put'.

Diese und alle anderen Zugriffe haben das allgemeine Format:

<ident>(<subscript>)

wobei

<ident> der Name des anzusprechenden Arrays ist.
<subscript> die Nummer des Elementes und ein arithmetischer Ausdruck ist.

Wie eben im Kommentar zur FOR-Schleife erwähnt, beginnen Arrays nicht bei Element 1, sondern bei Element 0. Das erste Element im Array 'cat' ist daher 'cat(0)' und nicht 'cat(1)'. Doch daran gewöhnt man sich schnell.

Beispiel 2

```
PROC changearray()

  BYTE ARRAY barray

  barray="Dies ist String 1."
  PrintC(barray)           ;gibt die CARD aus, die 'barray'
                           ;enthält
  Print(" ")
  PrintE(barray)           ;gibt den String aus, auf den
                           ;'barray' zeigt
  barray="Dies ist String 2."
  PrintC(barray)
  Print(" ")
  PrintE(barray)
RETURN
```

Ausgabe von Beispiel 2:

```
7663 Dies ist String 1.
7725 Dies ist String 2.
```

Kommentar zu Beispiel 2: Man beachte in der Bildschirmausgabe, dass sich die Adresse, auf die 'barray' zeigt, ändert. Die hier gezeigten Adressen sind abhängig vom eigenen ATARI-System und sehr wahrscheinlich andere, wenn man das Beispiel selbst ausprobiert.

Wird das Array mit einer String-Konstante neu belegt, wird der neue String nicht in die Speicherstellen des alten Strings geschrieben. Stattdessen wird für den neuen String ein neuer Speicherbereich bereitgestellt. Dann wird der Wert von 'barray' geändert, um auf die Startadresse des neuen Strings zu zeigen. Der alte String bleibt im Speicher erhalten, aber es ist kein Pointer mehr darauf gesetzt. Daher kann auf den alten String nicht mehr zugegriffen werden.

Merke, dass "PrintE(barray)" zulässig ist, weil 'barray' auf eine gültige String-Konstante zeigt, die von dem Parametertyp ist, den die Library PROC PrintE erwartet. Ziemlich raffiniert, nicht wahr?!

Beispiel 3

```
PROC equatearrays()

  BYTE ARRAY a="Dies ist eine String-Konstante!",
             barray
  barray=a
  PrintE(a)
  PrintE(barray)
RETURN
```

Ausgabe von Beispiel 3:

```
Dies ist eine String-Konstante!
Dies ist eine String-Konstante!
```

Anmerkung zu Beispiel 3: Dieses Programm zeigt, wie man zwei gleiche Arrays einfach dadurch erhält, dass einfach Zeiger auf die gleiche Speicherstelle gesetzt werden. In diesem Fall wird beides Mal auf die gleiche String-Konstante gezeigt.

Vielleicht ist schon aufgefallen, dass unterlassen wurde, dies zu konstruieren:

```
     BYTE ARRAY a= ['A '  's 't 'r 'i 'n 'g]
     PrintE(a)
```

Es würde nämlich nicht funktionieren. Man bedenke, dass sich String-Konstante von einfachen Strings dadurch unterscheiden, dass sie im ersten Byte die Länge des Strings speichern. Deshalb funktionieren PROCs, die eine String-Konstante erwarten, nicht mehr, wenn versucht wird, etwas anderes zu übergeben.

Und nun zu einem Programm, das alle gezeigten Möglichkeiten der Arrays nutzt.

Beispiel 4

Vorgaben: Man verfügt über ein Programm, das im Falle eines Bedienerfehlers nur Fehlernummern ausgibt. Und nun soll es auch Fehlermeldungen ausgeben. Das lässt sich mithilfe von Arrays erreichen, wie z.B. im nachfolgenden Programm. Wie es genau funktioniert, wird im Anschluss an das Programm erklärt.

```
PROC doerror(BYTE errnum)
;*** Diese PROC liest die Fehlernummer und gibt die
; dazugehoerige Fehlermeldung aus. Weitere Erlaeuterungen
; im Anschluss.

  BYTE ARRAY errmsg      ;die auszugebende Meldung speichert
  CARD ARRAY addr(6)     ;die Adressen der Fehlermeldungen
  addr(0)="Unzulaessiger Befehl"
  addr(1)="Unzulaessiges Zeichen"
  addr(2)="Falscher Dateiname"
  addr(3)="Zahl zu gross"
  addr(4)="Falscher Datentyp"
  addr(5)="Unbekannter Fehler"
  errmsg=addr(errnum)    ;schreibt die zu 'errnum'
```

```
  Print("Fehler #")       ;gehoerende Fehlermeldung in
  PrintB(errnum)          ;'errmsg' und gibt sie nach der
  Print(": ")             ;Fehlernummer aus
  PrintE(errmsg)
  PutE()
RETURN                    ;Ende der PROC doerror

PROC main()
;*** Diese PROC dient nur dazu, die obige PROC
; aufzurufen und dabei alle gueltigen Fehlernummern zu
; zeigen, um zu beweisen, dass die Tabelle korrekt ist.

  BYTE error
  FOR error=0 to 5
    DO
      doerror(error)
    OD
RETURN     ;*** Ende der Hauptprozedur
```

Ausgabe des Beispiels 4:

```
Fehler #1: Unzulaessiger Befehl
Fehler #2: Unzulaessiges Zeichen
Fehler #3: Falscher Dateiname
Fehler #4: Zahl zu gross
Fehler #5: Falscher Datentyp
Fehler #6: Unbekannter Fehler
```

Anmerkung zu Beispiel 4: Die Art und Weise, wie das CARD-Array in diesem Beispiel gesetzt wurde, ist neu. Wie kann man das Element eines CARD-Arrays mit einem String besetzen? Es ist aber vollkommen zulässig, weil nicht der String selbst einem Array-Element zugewiesen wird, sondern nur seine Adresse. Das macht jedes Element des Arrays zu einem unmittelbaren Pointer auf einen String. Was jetzt noch erforderlich ist, ist den Wert des momentanen Array-Elementes (z.B. das, das auf die benötigte Fehlermeldung zeigt) dem BYTE Array 'errmsg' zu übergeben. Dadurch wird 'errmsg' dazu gebracht, auf die jetzige Fehlermeldung zu zeigen. Danach wird die Fehlermeldung nur noch ausgegeben. Das Beispielprogramm mag ziemlich verwirrend erscheinen, falls das Konzept für Arrays und ihre interne Darstellung nicht völlig klar geworden ist. Trotzdem werden nachfolgend einige der fortgeschrittenen Fähigkeiten der Arrays aufgezeigt.

8.3 Records (Datensätze)

Records sind Konstruktionen, mit deren Hilfe Informationen, die zwar zusammengehören, aber nicht vom gleichen Datentyp sind, zusammengefasst werden können. Der Führerschein ist ein Beispiel für einen Record. Er enthält von Namen, Foto, Adresse und Fahrerlaubnisnummer. Alle diese Informationen gehören zusammen und beschreiben den Inhaber in einer gewissen Weise. Trotzdem handelt es sich um unterschiedliche Datentypen. Der Name ist eine Zeichenkette (String), das Foto ein Bild, die Adresse besteht aus Zeichen und Zahlen, ebenso die Fahrerlaubnisnummer. Natürlich unterstützt Action! nicht alle diese Datentypen. Es können nur die Datentypen zusammengefasst werden, die der Compiler auch verarbeiten kann: die elementaren Datentypen.

8.3.1 Records deklarieren

Action!-Records manipulieren die elementaren Datentypen insofern, als sie durch die Verbindung mit einem oder mehreren elementaren Datentypen neue Datentypen entwickeln. Danach werden zu dem neuen Datentyp Variablen deklariert, wie schon von der Variablendeklaration der BYTE-, CARD- oder INT-Typen bekannt ist. Dadurch ist es möglich, beliebig viele Variablen eines Record-Typs zu deklarieren, ohne das Format des Record-Typs jedes Mal neu festlegen zu müssen.

In Abschnitt 8.3.1.1 wird die Entstehung eines Record-Daten-Typs gezeigt und unter 8.3.1.2 die Deklaration von Variablen zu dem vordefinierten Record-Typ demonstriert.

8.3.1.1 Die Deklaration des Record-Typs

Ohne weitere Vorworte hier das allgemeine Format für die Deklaration eines Record-Daten-Typs:

TYPE <ident>=[<var decls>]

wobei

TYPE das Schlüsselwort ist,
<ident> der Name des Record-Typs ist,
<var decls> zulässige Deklarationen von Variablen sind[65].

Dazu noch ein hilfreiches Beispiel:

```
TYPE rec= [BYTE b1,b2      ;zuerst zwei BYTE-Felder,
           INT i1          ;dann ein INT-Feld
           CARD c1,c2,c3   ;dann drei CARD-Felder
           BYTE b3]        ;zum Schluss ein BYTE-Feld
```

Das Beispiel soll nun genau Stück für Stück erklärt werden:

TYPE rec	Deklaration eines neuen Datentyps mit Namen 'rec'.
BYTE b1,b2	Die ersten beiden Felder dieses Typs sind vom Datentyp BYTE und mit 'b1' und 'b2' bezeichnet.
INT i1	Das dritte Feld ist ein INT-Typ und heißt 'i1'.
CARD c1,c2,c3	Feld vier bis sechs sind CARD-Typen mit Namen 'c1, c2 und c3'.
BYTE b3	Das siebte und letzte Feld des Record-Typs 'rec' ist ein BYTE-Typ und heißt 'b3'.

Man beachte, dass zwischen den verschiedenen Variablendeklarationen keine Kommata stehen. Setzt man hier Kommas, versucht der Compiler die Worte für die elementaren Datentypen (CARD, BYTE, INT) als Variablen zu lesen, was natürlich einen Compiler-Fehler bedingt.

8.3.1.2 Variablen deklarieren

Im vorherigen Teil wurde gezeigt, wie ein Record-Typ deklariert wird. In diesem Abschnitt wird erklärt, wie Variablen eines vorgegebenen Record-Typs deklariert werden.

---

65) Kapitel IV, Abschnitt 3.4.1; Ausnahme davon ist die '=<init info>', die verboten ist!

Das Format ist der Variablendeklaration von elementaren Typen sehr ähnlich, weist aber einige Eigenheiten auf:

<ident> <var>{=<addr>}l:,<var>={=<addr>}:l

wobei

<ident>	der Name des Record-Typs ist.
<var>	eine Variable ist, deren Datentyp als Record-Typ deklariert werden soll.
<init info>	eine Information zum Setzen einiger Attribute dieser Variablen ist.
<addr>	die Speicherstelle ist, an der Sie die Variable setzen wollen. Es muss eine numerische Konstante sein.

Hier erst mal ein Beispiel für die Anwendung so eines Record-Typs. Nach dem Beispiel folgen die Erläuterungen dazu.

```
TYPE rec=[BYTE b1,b2    ;die Typ-Deklaration
INT i1                  ;aus dem vorherigen
CARD c1,c2,c3           ;Abschnitt, mit
BYTE b3]                ;einem BYTE endend

rec arec,               ;deklariert arec als Datentyp 'Record'
brec=$8000              ;deklariert brec als Typ 'Record' und
                        ;setzt ihn in Adresse $8000
```

Erklärungen:
'rec' gibt an, dass die nachfolgenden Variablen vom Typ 'rec' sind, wie z.B. BYTE, INT und CARD (wenn sie in Variablendeklarationen verwendet werden) anzeigen, dass die folgenden Variablen dieses Typs sind.

'arec' deklariert 'arec' als Variable vom Typ 'rec'.
'brec=$8000' deklariert 'brec' als Typ 'rec' und setzt sie auf Adresse $8000.

Nachdem nun klar ist, wie Variablen des Record-Typs deklariert werden und einige Daten dieses Typs bereits deklariert wurden, ist es Zeit, sich mit Zugriff auf und Manipulation von Records zu beschäftigen.

8.3.2 Record Manipulation

Sollen Records manipuliert werden, muss man wissen, wie auf ein Feld innerhalb eines Records zugegriffen wird. Das folgende Programm erledigt genau das unter Benutzung des Operators ('.'). Die Benutzung dieses Operators wird im Anschluss erklärt.

Beispiel

```
PROC recordreference()
;*** Diese Procedur liest einige Informationen ueber einen
;    Mitarbeiter ein, gibt sie dann zur Kontrolle wieder aus.

  TYPE idinfo=[BYTE level        ;Berechtigung
               CARD idnum,       ;Personalnummer
```

```
                    entry_year]  ;Einstellungsjahr

  idinfo rec     ;deklariert 'rec' als Record–Typ 'idinfo'

  Print("Ihre Personalnummer? ")
  rec.idnum=InputC()
  Print("Ihre Berechtigung (A-Z)? ")
  rec.level=GetD(7)
  PutE()
  Print("Ihr Einstellungsjahr? ")
  rec.entry_year=InputC()
  PrintE("O.K.! Das haben Sie eingegeben:")
  PutE()
  Print("Personalnummer # ")
  PrintCE(rec.idnum)
  Print("Berechtigung: ")
  Put(rec.level)
  PutE()
  Print("Einstellungsjahr: ")
  PrintCE(rec.entry_year)
RETURN ; ENDE
```

Ausgabe:

```
Ihre Personalnummer? 4365
Ihre Berechtigung (A-Z)? L
Ihr Einstellungsjahr? 1983
O.K.! Das haben Sie eingegeben:
Personalnummer # 4365
Berechtigung: L
Einstellungsjahr: 1983
```

Der '.' wird benötigt, um dem Compiler anzuzeigen, dass ein Zugriff auf einen Record erfolgen soll (nur hier zulässig!). Aus dem Beispiel kann man das allgemeine Format eines Record-Zugriffs ablesen:

<record name>.<field name>

Man beachte, dass <field name> und <record name> in verschiedenen Deklarationen definiert werden. <field name> wird in der TYPE-Deklaration festgelegt, wo ja auch die Felder eines Record-Typs bestimmt werden. Der <record name> dagegen wird in der Variablen-Deklaration bestimmt, wo die Variable als Record-Typ definiert wird.

8.4 Fortgeschrittene Anwendung erweiterter Datentypen

Die erweiterten Datentypen scheinen sehr beschränkt in der Anwendung zu sein, da sie nur elementare Datentypen verarbeiten können. Das liegt daran, dass es keine Arrays aus Records, Array Fields in Records usw. gibt. Aber es gibt einen Weg das zu umgehen, wie Beispiel 4 in Abschnitt 8.2.3 zeigt. In dem Beispiel wurde ein Array von Pointern eingerichtet, indem die Elemente eines CARD-Arrays als Pointer anstelle von CARD-Zahlen verwendet wurden. In diesem Abschnitt werden einige andere Möglichkeiten demonstriert, um mehr aus den erweiterten Datentypen herauszuholen. Dazu folgt auch ein Programm, das Records mit Array Fields verwendet, sowie ein Programm, das ein Array aus Records verwendet.

Grundsätzlich ist das natürlich überhaupt nicht zulässig! Das gilt aber nur, wenn man es auf direktem Wege versucht. Aber es wurde ja schon aufgezeigt, dass es so einige indirekte Möglichkeiten gibt. Das folgende Beispiel füllt ein dimensioniertes Array mit einer Liste von Records. Das funktioniert deshalb, weil dazu einfach ein "virtueller Record" definiert wird. Das Array ist in Wahrheit ein BYTE Array, bestehend aus Blöcken von Bytes, die in virtuelle Records gruppiert sind.

Ein virtueller Record ist kein Record im Sinne des Record-Typs. Er ist nur deshalb ein Record, weil auf einen Speicherbereich so zugegriffen wird, als ob es ein Record wäre - obwohl es sich in Wahrheit nur um eine Anzahl von Bytes handelt. Was hier passiert, ist ein BYTE-Array so zu füllen, dass es wie ein zusammenhängender Record aussieht und nicht wie Bytes. Das wird durch die Deklaration eines Record-Datentyps erledigt, auf den dann ein Pointer definiert wird. Dann wird das Array in Blöcken von der Größe eines Records dadurch manipuliert, dass der Pointer in Sprüngen von Recordgröße (in Bytes!) bewegt wird. Nach dem Beispielprogramm wird das genauer erörtert.

Beispiel 1

```
MODULE                       ; Deklaration globale Variablen
  TYPE idinfo= [CARD idnum,   ; Personalnummer
                     codenum, ; Zugangscode
                BYTE level]   ; Berechtigung
  BYTE ARRAY idarray(1000)    ; Speicher fuer 200 Records

  DEFINE recordsize="5"
  CARD reccount= [0]

PROC fillinfo()
  ;*** Die PROC nimmt Informationen auf, setzt sie per
  ;  Pointer in einen Record-Typ und indiziert den Pointer
  ;  innerhalb des Arrays. Dieser Vorgang wird so lange
  ;  fortgesetzt, wie neue Informationen eingeben werden.

  idinfo POINTER newrecord

  BYTE continue

  DO
    newrecord=idarray+(reccount*recordsize)
    Print("Personalnummer? ")
    newrecord.idnum=InputC()
    Print("Berechtigung (A-Z)? ")
    newrecord.level=GetD(7)
    PutE()
    Print("Zugangscode? ")
    newrecord.codenum=InputC()
    reccount==+1
    PutE()
    Print("Neuen Datensatz eingeben (J/N)? ")
    continue=GetD(7)
    PutE()
```

```
    UNTIL continue='N OR continue='n
  OD
RETURN
```

Programmieranmerkung: Diese PROC ist nicht auf das Array beschränkt! Auch Action! selbst prüft dies nicht! Der Programmierer muss durch eine Prüfroutine selbst dafür sorgen, dass die Grenzen des Arrays eingehalten werden!

Anmerkungen zum Beispiel: Diese PROC erledigt doch so einiges, was nun noch näher erklärt werden muss. Besonders die folgenden Zeilen:

```
DEFINE recordsize="5"
idinfo POINTER newrecord
newrecord=idarray+(reccount*recordsize)
newrecord.XXX=xxx
reccount==+1
```

Schrittweise eins nach dem andern. Es werden nicht nur die Anweisungen selbst, sondern auch die Konzeption erläutert, die verwendet wurde, um dieses aus Records bestehende Array zu erzeugen.

```
DEFINE recordsize="5"
```

Mit DEFINE wird hier die benötigte Sprungweite innerhalb des Arrays festgelegt. Der Record-Typ 'idinfo' ist 5 Bytes lang (2 CARDs und 1 BYTE), weshalb man sich in 5-Byte-Sprüngen durch das Array bewegen kann. Mit jedem Sprung bewegt man sich so um einen Record weiter. Selbst das nur teilweise Überschreiben eines bereits vorhandenen Records wird so verhindert.

```
idinfo POINTER newrecord
```

Hier wird ein Pointer auf den Typ 'idinfo' definiert. Die Felder eines virtuellen Records in einem Array kann man einfach dadurch füllen, dass man den Pointer auf das erste Feld eines virtuellen Records setzt. Und dann nutzt man den Pointer zum Zugriff auf ein einzelnes Feld innerhalb des Records.

```
newrecord=idarray+(reccount*recordsize)
```

Durch diese Zuweisung zeigt der Pointer auf die nächste freie Speicheradresse nach dem Ende des Arrays. Das wird durch Addition des für die Records verbrauchten Speichers zur Anfangsadresse des Arrays erreicht. Der belegte Speicher wird einfach berechnet durch Anzahl der Records ('reccount') mal Größe des Records ('recordsize').

```
newrecord.XXX=xxx
```

'XXX' ist einer der Feldnamen des Records, 'xxx' ist die dazu gehörige Eingabe, die in das Array geschrieben wird. Da 'newrecord' auf das Ende des Arrays zeigt, kann man direkt den neuen Record füllen. Den Pointer aus dem Recordzugriff kann man deshalb benutzen, weil er als Pointer auf diesen Record-Typ definiert wurde.

```
reccount==+1
```

Hier wird nur die Variable hochgezählt, welche die Anzahl der im Array vorhandenen Records beinhaltet. Das ist für neue Eingaben erforderlich.

Dieses Array wird später im Beispiel 4 verwendet, um die Eingaben eines Benutzers zu verifizieren, der eine Sperrzone betreten will. Dabei darf man nicht vergessen, dass auf das Array als ein Array aus Records zugegriffen wird, wobei das für die Eingabe definierte Format verwendet wird. Andernfalls könnten merkwürdige Dinge passieren.

Bevor es mit dem Programm weitergeht, das auf die besetzten Arrays zugreift, werden die Records ein wenig modifiziert. Es wird ein Feld hinzugefügt, das den Namen des Benutzers im Format

```
Nachname, Vorname
```

aufnimmt. Um das zu erreichen, muss daraus ein Array erstellt werden. Aber wie?

Es wird einfach ein BYTE-Field an das Ende des Record-Typs angehängt und die Direktive DEFINE auf die neue Record-Größe abgeändert. Wird die Größe um 20 Bytes erhöht, werden nun für 6 "Field Bytes" (2 CARDs und 2 BYTEs) 25 Bytes Speicher reserviert. Nun wird der String durch Zugriff auf das letzte Feld in den Extraspeicher gepackt (unser neues BYTE Field), indem dann Strings statt Bytes eingegeben werden. Der String kann maximal 19 Zeichen fassen (das 1. Zeichen jedes Strings in Action! gibt ja seine Länge an!), deshalb muss die Eingabe später auf 19 Zeichen beschränkt bleiben. Ohne weitere Umschweife hier die abgeänderte Version der PROC 'idinfo'.
Beispiel 2

```
MODULE          ;Deklaration globaler Variablen

  TYPE idinfo= [CARD idnum,     ; Personalnummer
                     codenum,    ; Zugangscode
                BYTE level,     ; Berechtigung
                     name]      ; erster Buchstabe des Namens

  BYTE ARRAY idarray(1000) ; genug Speicher für 40 Records

  DEFINE recordsize="25",
         nameoffset="5"
  CARD reccount=[0]

PROC fillinfo()
;*** modifizierte Version des vorherigen Beispiels 1

  idinfo POINTER newrecord

  BYTE POINTER nameptr           ; Pointer auf Namenfeld
  BYTE continue

  DO
    newrecord=idarray+(reccount*recordsize)
    Print("Personalnummer? ")
    newrecord.idnum=InputC()
```

```
    Print("Berechtigung (A-Z)? ")
    newrecord.level=GetD(7)
    PutE()
    Print("Zugangscode? ")
    newrecord.codenum=InputC()
    nameptr=newrecord+nameoffset
    PrintE("Name? ")
    Print("(Form: Name, Vorname) ")
    InputS(nameptr)
    reccount==+1
    PutE()
    Print("Neuen Datensatz eingeben (J/N)? ")
    continue=GetD(7)
    PutE()
    UNTIL continue='N OR continue='n
  OD
RETURN
```

Anmerkungen zu Beispiel 2:

Diesmal müssen folgende Zeilen genau betrachtet werden.

```
nameoffset="5"

BYTE POINTER nameptr

nameptr=newrecord+nameoffset

inputS(nameptr)
```

Bevor die Zeilen im Einzelnen erläutert werden, wird kurz die Methode beleuchtet, wie man einen Namen in ein Array aus Records hineinsetzt. Zuerst gilt es herauszufinden, wohin der Name nach der Eingabe geschrieben werden soll.

```
nameoffset="5"
```

Damit wird der Abstand 'DEFINEd', mit dem man in den einzelnen Record hineingehen muss, um das erste Byte des Strings zu bekommen. Dadurch zeigt der Pointer für den String auf die korrekte Position.

```
BYTE POINTER nameptr
```

Dieser Pointer zeigt auf das erste Byte des 'Namen-Feldes' innerhalb des Records.

```
nameptr=newrecord+nameoffset
```

Hier wird der Wert des Pointers 'nameptr' gesetzt (wohin er z.B. zeigen soll). Man addiert zur Startadresse des Records ('newrecord') das Offset auf die Speicheradresse des ersten Bytes von dem gespeicherten String.

```
InputS(nameptr)
```

Dient zum Einlesen des Namens und verwendet 'nameptr' als Pointer auf die Speicheradresse, wie schon in Abschnitt 8.2.3 (Beispiel 2) gezeigt. Allerdings mit der Ausnahme, dass nun ein Pointer anstelle eines Array-Namens benutzt wird, der dann auf das erste Element zeigt.

Nachdem jetzt klar ist, wie die Records in dem Array gespeichert werden, fehlt noch ein Weg, um das Array nach einem bestimmten Record durchsuchen zu können. Die nachfolgende Funktion wurde dafür ausgelegt. Sie greift auf das Array zu, benutzt dabei das Record-Format aus Beispiel 2 und liefert den Anfang des ersten Records, der eine 'idnum' enthält, die dem entsprechenden Suchparameter entspricht. Wird nichts gefunden, wird eine 0 als Adresse ausgegeben. Beachtenswert ist, dass diese Funktion Variablen benutzt, die in der globalen Anweisung des früheren Beispiels deklariert wurden (also nach MODULE).

Beispiel 3

```
CARD FUNC findmatch(CARD testidnum)

  idinfo POINTER seeker ; zeigt waehrend des Testes
                        ; auf jeden Record

  BYTE ctr              ; Zaehler in der FOR-Schleife

  FOR ctr=0 TO (reccount-1) ; minus 1, da Start bei, nicht 1!
  DO
    seeker=idarray+(ctr*recordsize) ; Index fuer den Record
    IF seeker.idnum=testidnum THEN  ; Suche nach Treffer
      RETURN (seeker)               ; Ruecksprung bei Fund
    FI
  OD
RETURN (0);                   nichts gefunden, Ende der FUNC
```

Eine kleine Erläuterung zur FUNC. Alles was hier passiert, ist der Test jedes Feldes 'idnum' auf die Zahl 'testidnum'. Jetzt werden die letzten beiden Beispiele in ein wirkliches Programm verwandelt, indem eine richtige Struktur verwendet wird.

Beispiel 4

```
MODULE                ; Deklaration einiger globaler Variablen

  TYPE idinfo= [CARD idnum,     ; Personalnummer
                     codenum    ; Zugangscode
                BYTE level,     ; Berechtigung
                     name]      ; erster Buchstabe des Namens

  BYTE ARRAY idarray(1000) ; Speicher fuer 40 Records

  DEFINE recordsize="25",
         nameoffset="5"

  CARD reccount=[0]
```

```
PROC fillinfo()            ; *** die bekannte Eingaberoutine

  idinfo POINTER newrecord
  BYTE POINTER nameptr     ; Pointer auf das 'name'-Feld
  BYTE continue
  DO
    newrecord=idarray+(reccount*recordsize)
    Print("Personalnummer? ")
    newrecord.idnum=InputC()      ; Eingabe Personalnummer
    Print("Berechtigung (A-Z)? ")
    newrecord.level=GetD(7)       ; Eingabe Berechtigung
    Put(newrecord.level)
    PutE()
    Print("Zugangscode? ")
    newrecord.codenum=InputC()    ; Eingabe Zugangscode
    nameptr=newrecord+nameoffset  ; 'nameptr' auf den Start
                                  ; des Namenfeldes setzen
    PrintE("Name? ")
    Print("(Form: Name, Vorname) ")
    InputS(nameptr)   ; Namen in das Namenfeld einlesen
    reccount==+1
    PutE()
    Print("Einen weiteren Record eingeben (J/N)? ")
    continue=GetD(7)
    PutE()
    UNTIL continue='N OR continue='n
  OD
RETURN

CARD FUNC findmatch(CARD testidnum)

  idinfo POINTER seeker ;zeigt auf jeden Record zum Vergleich
  BYTE ctr              ;Zaehler in der FOR-Schleife

  IF reccount>0 THEN      ; Endlosschleife verhindern,
                          ; falls noch keine Daten eingegeben
    FOR CTR=0 TO (reccount-1)
      DO
        seeker=idarray+(ctr*recordsize) ; Index fuer Record
        IF seeker.idnum=testidnum THEN  ; Suche nach einer
          RETURN (seeker)               ; vorhandenen Perso-
        FI                             ; nalnummer und Rueck-
                                       ; sprung falls gefunden
      OD
  ELSE
    PrintE("Noch keine Daten eingegeben!")
  FI
RETURN (0);nichts gefunden, Ende der FUNC findmatch

PROC main()
;*** diese PROC kontrolliert den gesamten Programmablauf
```

```
idinfo POINTER recptr     ; Pointer auf einen Record

BYTE POINTER nameptr      ; Pointer auf 'name'-Feld

CARD id_num,        ; Eingabe Personalnummer durch Anwender
     code_num,      ; Eingabe Zugangscode durch Anwender
     keyid=[65535]  ; Ausstiegscode aus der Schleife

BYTE mode           ; steuert den Ablauf

Put(125)            ; Bildschirm loeschen
PrintE("Auf geht's...!")
PrintE("Welcher Verfahrensmodus?")
PrintE("X = Mitarbeiterliste ergaenzen")
PrintE("A = Alarm/Test des Eingabemodus")
Print(">> ")
mode=GetD(7)                    ; Modus eingeben und
Put(mode)                       ; auch anzeigen
PutE()
IF mode#'A AND mode#'a THEN      ; alles ausser A oder a
  fillinfo()       ; fuehrt in den X-Modus
ELSE               ; Abfrageroutine
  DO                            ; Schleife startet
    Print("Personalnummer >> ")
    id_num=InputC()             ; Eingabe Personalnummer
    IF id_num=keyid THEN        ; Ausstiegscode
      EXIT                      ; aus Endlosschleife
    ELSE                        ; normale Personalnummer
      recptr=findmatch(id_num)  ; Suche nach eingegebener
                                ; Personalnummer
      IF recptr=0 THEN          ; nix gefunden
        PrintE("Kein Zugang!")
      ELSE                      ; Personalnummer gefunden
        Print("Codenummer >> ")
        code_num=InputC()       ; Zugriffscode eingeben
        IF recptr.codenum=code_num THEN ; gefunden
          nameptr=recptr+nameoffset
          Print(" Personalnummer # ")   ; vorhandene
          PrintCE(recptr.idnum)         ; Informationen
          Print("Berechtigung: ")       ; auf
          Put(recptr.level)             ; dem
          PutE()
          Print("Name: ")               ; Bildschirm
          PrintE(nameptr)               ; ausgeben
          PutE()
          PrintE("O.K.! Passieren!")
        ELSE                    ; Zugangscode falsch
          PrintE("Kein Zugang!")
        FI         ; Ende Ueberpruefung des Zugangscodes
      FI           ; Ende Ueberpruefung der Personalnummer
    FI             ; Ende Ueberpruefung auf Schluesselcode
```

```
    OD               ; Ende der Endlosschleife
  FI                 ; Ende der IF-Abfragen ...
  Put(125)           ; Bildschirm loeschen
  PrintE( "System schaltet ab ... ")
RETURN               ; Ende der PROC main
```

Die PROC 'main' arbeitet nur eine Reihe Überprüfungen ab, um festzustellen, was bei jedem Programmpunkt auszuführen ist. Die verschachtelten IF irritieren ein wenig, sind aber durch Einrücken gleichmäßig ausgerichtet. So gelingt beim Lesen leicht die Zuordnung von IF-FI etc.

9 Fortgeschrittene Konzeptionen

Dieser Abschnitt zeigt Programmiertechniken auf, die erfahrene Programmierer als hilfreich bewerten mögen. Bisher wurden nur Erläuterungen zur Programmiersprache selbst ohne Bezug auf den Rest des Computersystems gegeben. Abschnitt 9 ist Informationen über den Zusammenhang von Action! mit dem ATARI-System gewidmet, einschließlich Betriebssystemroutinen und Systemvariablen.

9.1 Code-Blöcke

Damit lässt sich Maschinencode in ein Action!-Programm einbinden. Der Compiler übernimmt die Werte eines Blocks ohne Prüfung als bereits kompilierten Code. Daher sollten Code-Blöcke nur dann eingefügt werden, wenn Kenntnisse in Assembler und Maschinensprache bestehen.

Das allgemeine Format eines Code-Blocks lautet:

[<value>l: <value>:l]

wobei

<value> ein Wert aus dem Code-Block ist.

Es muss zwingend eine Compiler-Konstante sein[66]. Werte größer als 255 werden im LSB-MSB-Format gespeichert.

Beispiele

```
[$40 $0D $51 $F0 $0600]

BYTE b1,b2,b3
['A b1 342 b3 4+$A7]
DEFINE on=1
[54 on on+'t $FFF8]
```

Code-Blöcke sind ziemlich hilfreich bei der Einflechtung kurzer Maschinensprache-Routinen. Für lange Unterprogramme eignen sie sich nicht. Siehe dazu Abschnitt 9.4!

66) Kapitel IV, Abschnitt 3.2.

9.2 Adressieren von Variablen

In den vorherigen Abschnitten[67] wurde gezeigt, dass die Adresse einer Variablen durch ihre Deklaration bestimmt werden kann. Bisher wurde davon kein rechter Gebrauch gemacht und noch nicht einmal die Nützlichkeit dieser Option erläutert.

Mit dieser Option kann in Action! eine Variable deklariert werden, welche die gleiche Adresse wie ein beliebiges Hardware-Register besitzt! Damit lassen sich Sound und Grafik direkt beeinflussen, das Betriebssystem unmittelbar verändern usw. ...

Um die Vorteile zu verdeutlichen, folgt nun ein Grafikprogramm, das scrollt und die Hintergrundfarben verändert. Dafür sind die normalen Farbregister (Schattenregister) nicht geeignet, da sie bei jedem Bildaufbau (VBI) nur einmal gelesen werden.

Stattdessen werden die Hardware-Farbregister direkt geändert. Auf diese Weise kann die Hintergrundfarbe im laufenden Bild verändert werden. Tatsächlich ist das 12mal möglich, sodass man so 12 Farben in Graphics 0 bekommt. Allerdings muss man dafür Sorge tragen, dass die Farbe nicht in der Mitte der Bildzeile geändert wird. Daher wird die Systemvariable WSYNC benutzt, die mitteilt, wann eine Bildzeile (Scan Line) zu Ende gezeichnet ist und die Nächste noch nicht begonnen wurde. Die Variable VCOUNT gibt an, wie viele Scan Lines ausgegeben wurden, und wird deshalb zum Timing verwendet.

Beispiel

```
PROC scrollcolors()

  BYTE wsync=54282,        ; the "wait for sync" flag
       vcount=54283,       ; the "scan line count" flag
       clr=53272,          ; hardware register background
       ctr,chgclr=[0] ,    ; a counter and color changer
       incclr              ; increments color luminance
  Graphics(0)              ; set GR.0
  PutE()
  FOR ctr=1 to 23          ; print out demo message
    DO
    PrintE("A DEMO OF SHIFTING BACKGROUND COLORS")
    OD
  Print("A DEMO OF SHIFTING BACKGROUND COLORS")
  DO                       ; start of infinite scrolling loop
  FOR ctr=1 TO 4
    DO
    incclr=chgclr          ; set base color to increment
      DO                   ; start of until loop
      wsync=0              ; waits for end of scan line
      clr=incclr           ; change displayed color
      incclr==+1           ; change luminance
      UNTIL vcount&128     ; end of screen test
      OD                   ; end of UNTIL loop
    OD                     ; end of FOR loop
  chgclr==+1               ; change the base color
```

67) Kapitel IV, Abschnitte 3.4.1, 8.1.1 und 8.2.1.

```
  OD                         ; end of infinite scrolling loop
RETURN                       ; end of PROC scrollcolors
```

9.3 Adressieren von Routinen

Das Konzept zur Bestimmung der Adresse einer Routine ist ähnlich dem zur Bestimmung der Adresse einer Variablen. Im letzten Abschnitt wurde die direkte Benutzung von ATARI-Systemregistern durch Adressierung einer Action!-Variablen auf das zuständige Register dargelegt. Da man die Adresse einer Routine bestimmen kann, lassen sich direkte Einsprünge in das Betriebssystem sowie die Hardware-Routinen vornehmen und ermöglichen so Manipulationen der Ein-/Ausgabe. Die verwendete Methode wird im nachfolgenden Abschnitt erklärt. Sie trifft auf alle Routinen in Maschinensprache zu, seien sie selbst geschrieben, im Betriebssystem oder im ROM vorhanden.

9.4 Maschinensprache und Action!

Action! ermöglicht den einfachen Aufruf von Routinen in Maschinensprache (MS). Es gilt dabei nur zwei Regeln zu beachten:

- Die Startadresse der Routine muss bekannt sein.
- Die Routine muss mit einem 'RTS' enden (dezimal 96), will man zu Action! zurück.

Für MS-Programmierer sicher kein Problem. Und Parameter? Klar kann man Parameter an die MS-Routinen übergeben! Der Compiler speichert Parameter wie folgt:

Adresse	n-tes Byte der Parameter
A-Register	1.
X-Register	2.
Y-Register	3.
$A3	4.
$A4	5.
:	:
:	:
$AF	16.

Beispiel

```
PROC CIO=$E456(BYTE areg,xreg)
; *** Deklarieren der OS-PROC CIO. 'xreg' enthaelt die
; IOCB-Nummer mal 16, und 'areg' ist nur ein Fuellbyte,
; weswegen die Zahl nicht in das Register A uebernommen wird
; (CIO erwartet den Wert im X-Register)

PROC readchannel2()
; *** Diese PROC oeffnet Kanal 2 auf den eingegebenen
; Dateinamen und ruft die CIO auf, um eine 'buflen' Anzahl
; an Bytes einzulesen
```

```
  DEFINE buflen="$1E78"     ; Laenge des Puffer-Arrays
  BYTE ARRAY filename(30),  ; das Array fuer Filename
       buffer(buflen)       ; das Puffer-Array

  BYTE iocb2cmd=$362   ; Kommandobyte für IOCB #2

  CARD iocb2buf=$364,  ; Startadresse des Puffers für IOCB #2
       iocb2len=$368   ; Laenge des Puffers für IOCB #2

  PutE()
  Print("Dateiname >> ")
  InputS(filename) ; Dateiname eingeben
  Open(2,filename,4,0) ;Kanal #2 zum Lesen oeffnen
  iocb2cmd=7 ; Kommando 'get binary record' (lade 'n' Bytes)
  iocb2buf=buffer ; den IOCB-Puffer auf unseren Puffer setzen
  iocb2len=buflen ; IOCB-Pufferlaenge setzen
  CIO(0,$20)      ; *** Aufruf der CIO ***
  Close(2)        ; Kanal #2 schliessen
RETURN
```

Ist das nicht einfach? So kann man eigene MS-Routinen in eine Hochsprache einbinden, in der die Entwicklung einer Programmstruktur leichter von der Hand geht.

9.5 Fortgeschrittener Einsatz von Parametern

In Abschnitt 6.4 wurden Parameter und ihre Verwendung erklärt, wobei auch erläutert wurde, dass man aus einer Routine heraus keinen Wert mithilfe eines Parameters ausgeben könne. Nun, das war eine kleine Notlüge. Mit Hilfe von Pointer-Angaben geht es nämlich doch!

Man muss nur einen Pointer einrichten, der auf die Variable zeigt, die man tatsächlich an eine Routine übergeben will und stattdessen den Pointer übergeben. Wenn man dann auf das zugreift, worauf der Pointer zeigt, greift man tatsächlich auf die Variable zu, die man eigentlich übergeben wollte. So kann dann der Wert der Variablen durch Verwendung von Pointer-Angaben leicht verändert werden.

Dies ist eine indirekte Methode (hier: benutzen eines Pointers auf eine Variable statt der Variablen), die aber trotzdem sehr effizient ist und in einigen Fällen äußerst nützlich sein kann, wie das folgende Beispiel zeigt.

Beispiel

```
BYTE FUNC substr(BYTE ARRAY str,sub
                 BYTE POINTER errptr,notfound)

; *** Diese FUNC durchsucht 'str' nach dem Substring 'sub'.
; Wird dieser gefunden, liefert die Funktion den Index auf
; den String zurueck. Ist 'substr' laenger als 'str', wird
; ueber 'pointer' ein Fehler ausgegeben. Wird 'substr' nicht
; gefunden, wird das ueber einen anderen Pointer zurueck-
; gemeldet.
```

```
  BYTE ARRAY tempstr ; enthaelt temporaeren Substring fuer
                     ; den Test
  BYTE ctr1,         ; Zaehler fuer die aeussere Schleife
       ctr2          ; Zaehler fuer die innere Schleife

  IF sub(0)>str(0) THEN   ; 'substr' ist groesser als 'str'
    errptr^=1
  ELSE
    FOR ctr1=1 TO str(0)  ; Schleife fuer Test des Strings
      DO
      IF sub(1)=str(ctr1) THEN     ; teste ersten Buchstaben
         tempstr(0)=sub(0)         ; dimensioniere tempstr
        FOR ctr2=1 TO sub(0)       ; fuelle tempstr
          DO
          tempstr(ctr2)=str(ctr2+ctr1-1)  ; fuelle tempstr
          OD
        IF SCompare(tempstr,sub)=0 THEN   ; Vergleich
                                          ; 2er Strings
           RETURN (ctr1) ; return index if equal
        FI
      FI                  ; Ende des Tests von Zeichen 1
      OD                  ; Ende der FOR-Schleife
  FI
  notfound^=1             ; kein Ergebnis aus der Schleife
                          ; also keine Entsprechung gefunden
RETURN (0)                ; Ende der FUNC substr
```

Der Aufruf dieser Routine erfolgt in der Form:

<index>=substr(<string>,<substring>,<errptr>,<nofindptr>)

wobei

<index>	der Index in <string> hinein ist, wo <substring> beginnt.
<string>	der Hauptstring ist.
<substr>	der substr ist, der im main string gefunden werden soll.
<errptr>	der Zeiger auf das Byte 'Error Flag' ist.
<nofindptr>	ein Zeiger auf das Byte 'substring notfound' ist.

Diese Art der Manipulation von Parametern braucht etwas Übung, falls man mit der Benutzung von Pointern nicht vertraut ist. Aber es ist ein schneller und einfacher Weg, mehr Informationen aus einer Routine zu erhalten, ohne mit globalen Variablen arbeiten zu müssen. Dadurch bleibt eine Routine auch wirklich eine Mehrzweckroutine, wie in Abschnitt 6.4 beschrieben.

V. Der Action!-Compiler

1 Einführung

ATARI BASIC bietet ein hohes Maß an Bequemlichkeit, da Programme in einer Art Englisch geschrieben werden und dann sofort ohne weitere Arbeitsschritte ausprobiert werden können. Dieser doppelte Vorteil wird durch einen hohen Aufwand erkauft. Jeder Befehl in jeder Zeile muss einzeln durch ein spezielles Programm (BASIC-Interpreter) während des Programmablaufs übersetzt und ausgeführt werden.

Action! ist da etwas ausgefeilter. Ein Programm wird zuerst durch den Compiler vollständig übersetzt, bevor es ausgeführt wird. Das verlangt einen Zwischenschritt zwischen der Eingabe des Programms und seiner Ausführung durch den Computer. Dieser Vorgang wird "Kompilierung" genannt. Während des Kompilierens analysiert der Compiler das Programm zeilenweise und übersetzt es dann in eine andere Computersprache (Maschinensprache), wobei sowohl globale als auch lokale Variablen separat abgespeichert werden. Das konvertierte Programm wird dann vom ATARI ausgeführt, wobei die Geschwindigkeit sehr viel höher als bei interpretiertem ATARI BASIC ist.

Da der Editor mit zwei Fenstern[68] arbeitet, kann man aus dem jeweilig aktiven Fenster heraus kompilieren. Egal, von welchem Fenster aus der Compiler angesprochen wurde, nach dem Rücksprung zum Editor landet man immer in Fenster #1. <CTRL><SHIFT> <2> führt in Fenster #2.

1.1 Der Sprachumfang

Dieses Kapitel nimmt Bezug auf einige bereits erläuterte Ausdrücke[69]. Zur Erinnerung:

Ausdruck	Erläuterung
<ident>	jede zulässige Kennung
<value>	jeder zulässige Dez-/Hex-Wert
<compiler constant>	berechnet <ident>'s Adresse
<address>	Speicherstelle

1.2 Compiler – Direktiven

Direktiven wurden bereits ausführlich erläutert[70]. Hier nochmals der Hinweis: Compiler-Direktiven werden nur während der Kompilierung ausgeführt, nicht aber beim Programmablauf. Also keine Compiler-Direktiven für den Programmablauf einsetzen.

2 Speicheraufteilung/-Zuweisung

Der Compiler weist Speicherplatz für das kompilierte Programm, die Variablen, die Routinen und die Symboltabelle wie folgt zu. Nach dem Aufruf liest der Compiler als Erstes Speicherstelle 14. Der CARD-Wert, der sich aus den Inhalten dieser Speicherstelle plus der folgenden ermittelt, ergibt den Beginn des freien Speichers. Diese Adresse kann in Abhängigkeit vom Editor-Puffer[71] variieren. Solange das nicht geändert wird, legt der Compiler beginnend ab dieser Speicherstelle das kompilierte Programm ab. Soll das kompilierte Programm in einem anderen Speicherbereich abgelegt werden,

68) Kapitel II, Abschnitt 2.6
69) Kapitel IV, Abschnitt 2.1
70) Kapitel IV, Abschnitt 7
71) Speicheraufteilung in Anhang B

gibt man unmittelbar vor dem Kompilieren die folgenden zwei Kommandos an den Monitor:

```
SET 14=<address>
SET $491=<address>
```

wobei <address> die Startadresse für das kompilierte Programm ist.

Anmerkung: Wenn eine Adresse angeben wird, die unterhalb des Endes vom Quelltext im Speicher liegt, kann eventuell der Quelltext im Editor überschrieben werden. Daher empfiehlt sich bei dieser Vorgehensweise das Kompilieren von einem Speichergerät[72].

2.1 Kommentare, SET, DEFINE

Weder Kommentare noch die Direktiven SET oder DEFINE erzeugen Maschinencode. Da sie beim Programmablauf keine Funktion haben, werden sie nicht benötigt.

2.2 Zuweisung von Variablen

Informationen über Variablen werden vom Action!-Compiler an zwei verschiedenen Stellen abgelegt - im Code selbst und in der Symboltabelle. Die Symboltabelle wird später betrachtet.

Die Variablen werden direkt vor dem Maschinencode abgespeichert, von dem sie benötigt werden. Einige Variablen sind bereits deklariert, bevor die erste Routine angesprochen wird. Diese sogenannten globalen Variablen können von jeder nachfolgenden Routine verwendet werden. Sie müssen innerhalb der Routinen nicht mehr erneut deklariert werden. Für die zugewiesenen Variablen wird Speicherplatz reserviert ähnlich wie bei der Definition der elementaren Datentypen. Die Tabelle verdeutlicht das:

Datentyp	Zuweisung	Kommentar
BYTE	1 Byte	elementarer Typ
CHAR	1 Byte	elementarer Typ
CARD	2 Bytes	elementarer Typ
INT	2 Bytes	elementarer Typ
ARRAY	Größe mal Anzahl der Elemente	erweiterter Typ
TYPE	Summe der Größen elementarer Typen, hängt ab von Deklaration	erweiterter Typ
string	alle Zeichen im String plus ein vorangestelltes Byte für die Länge	jeder String wird extra abgelegt, sogar wenn er dem gleichen <ident> zugewiesen wurde

2.3 Routinen

Der Compiler reserviert Speicherplatz für Routinen (PROCs und FUNCs) im Anschluss an den Bereich, der für die Deklaration der globalen Variablen festgelegt wurde. Die lokalen Variablen werden in der betreffenden Routine vorangestellt. Dann folgt der in Maschinencode übersetzte Programmtext.

72) Kapitel III, Abschnitt 2.2.

2.4 INCLUDEte Programme

Weitere Programme können an jeder Stelle im Programm eingefügt werden. Natürlich darf der eingefügte Programmtext nicht mit dem Text in Konflikt geraten, der gerade vom Compiler abgearbeitet wird. Besonders muss man dabei auf gleichlautende Kennungen und zusammenhanglose Einfügungen achten. Werden im INCLUDEten Text Fehler entdeckt, werden diese wie gewohnt angezeigt. Die Fehlernummer wird in der Kommandozeile des Monitors ausgegeben und es ertönt der Summer.

2.5 Zusätzliche globale Variablen - MODULE

Zusätzliche globale Variablen, Arrays und Records können bei Bedarf hinzugefügt werden, indem das Schlüsselwort MODULE verwendet wird. Den Variablen wird Speicherplatz im Anschluss an die letzte zuvor abgearbeitete Routine zugewiesen. Die Kennungen werden ebenfalls in die Symboltabelle des Compilers aufgenommen.

2.6 Symboltabellen

Der Compiler führt zwei Symboltabellen - eine für die globalen Variablen und eine für die lokalen Variablen aus der zuletzt kompilierten Routine. Auf die Symboltabellen kann per Monitor mithilfe der Kommandos '?', '*' und SET zugegriffen werden[73]. Die Tabellen werden auch dann benutzt, wenn der Compiler die Adresse einer Variablen benötigt.

Der Compiler reserviert für diese Tabellen 2 KBytes (8 Pages) am oberen Ende des freien Speichers (MEMTOP - $800). Aufgrund dieser Speicherreservierung ist es sehr leicht möglich, die Tabellen durch ein Programm zu überschreiben, das während des Ablaufs in einen Grafikmodus wechselt, der mehr Speicher als GRAPHICS 0 benötigt. Daraus folgt, dass man während der Programmausführung nicht zurück in den Monitor gehen und die Werte der verwendeten Variablen prüfen kann. Der Compiler wird sie nicht mehr finden, weil sie gerade überschrieben wurden. Bei Bedarf kann der für die Symboltabelle zu reservierende Speicher unmittelbar vor dem Kompilieren vergrößert werden. Ebenso kann die Anzahl der Symbole auf maximal 510 erhöht werden.[74]

3 Options-Menü nutzen

Das Options-Menü des Monitors bietet einige Möglichkeiten zur Erweiterung bzw. Veränderung des Kompiliervorgangs an[75].

3.1 Erhöhen der Kompiliergeschwindigkeit

Die Geschwindigkeit lässt sich um bis zu 30% Prozent erhöhen, wenn beim Kompilieren und bei I/O-Operationen der Bildschirm abgeschaltet wird. Dazu die Frage 'Screen?' im Optionsmenü mit 'N<RETURN>' beantworten.

3.2 Warnton abschalten

Der Warnton kann bei der Fehlersuche manchmal ganz schön nerven. Abschalten mit der Antwort 'N<RETURN>' auf die Frage 'Bell?'.

3.3 Unterscheidung von Groß- und Kleinschreibung

Sofern ein gehobener Programmierstil gepflegt werden soll, wird vielleicht Wert darauf gelegt, dass der Compiler die entsprechende Schreibweise der Schlüsselwörter prüft. Oder wird lieber die gemischte Schreibweise bevorzugt? Beides ist möglich. Wird die

---

73) Kapitel III, Abschnitte 2.8, 2.11 und 2.12.
74) Kapitel VII, Abschnitt 4.
75) Kapitel III, Abschnitt 2.5. Übersicht in Anhang F.

Frage 'Case sensitive?' mit 'Y<RETURN>' beantwortet, unterscheidet der Compiler Groß-/Kleinschreibung. Die Lesbarkeit der Quelltexte lässt sich damit verbessern und es werden mehr unterschiedliche Kennungen (<ident>'s) möglich. Aber dann müssen zwingend alle Schlüsselwörter[76] in Großbuchstaben geschrieben werden.

3.4 Auflisten des zu kompilierenden Programmtextes

Der Compiler kann während des Kompilierens jede zu übersetzende Zeile auf dem Bildschirm ausgeben. Das erscheint überflüssig, da ja die Masse der Fehler erkannt und angezeigt wird. Aber sobald andere Routinen oder Programme per INCLUDE eingefügt werden, wird es unübersichtlich. Und ein vollständiges Listing des gesamten Programms kann auch nicht angefertigt werden, was mit dieser Option jedoch möglich wird. Man kann die Ausgabe auf den Drucker umleiten[77], um ein komplettes Listing auf Papier zu bekommen. Dazu setzt man die Option 'List?' auf 'Y<RETURN>'.

4 Technische Betrachtungen

4.1 Overflow und Underflow

Der Compiler prüft nicht, ob ein mathematischer Überlauf bzw. das Gegenteil vorliegt.

Enthält eine BYTE-Variable den Wert 255 und wird 1 dazu addiert, erhält man nicht 256, sondern 0 (ein Byte kann nur max. 255 enthalten). Wird von 0 der Wert 1 subtrahiert, passiert das Gegenteil. Man erhält 255.

Wie bereits erwähnt[78], bedingen einige mathematische Operatoren bestimmte Ausgaben. Beachtet man diese, werden derartige Probleme vermieden.

Auch können Shift-Operationen Overflow bzw. Underflow bewirken. Das Shiften des Inhalts einer Variablen produziert ähnliche (aber nicht identische) Ergebnisse, wie die Multiplikation bzw. Division mit 2.

4.2 Prüfen von Typkompatibilität und Bereichsgrenzen

Gleichfalls muss berücksichtigt werden, dass der Compiler die Grenzen von einfachen Variablen oder Arrays nicht prüft. Das ist mit Vorbedacht so gestaltet worden, damit mehr Möglichkeiten zur Datenmanipulation zur Verfügung stehen. Der Preis für diese Freiheit ist erhöhte Aufmerksamkeit. Man muss eigene Prozeduren für Fehlerbehandlung und Prüfungen von Bereichsgrenzen entwickeln. Ein guter Grund für eine entsprechende Routinensammlung, aus der dann INCLUDEt werden kann.

4.3 Beschränkungen von IOCB #7

Beim Start öffnet das Action!-System den IOCB #7 für die Tastatur (K:). Dafür sollte IOCB auch reserviert werden. Man kann den Kanal für diesen Zweck in eigenen Programmen verwenden, sollte aber nicht die Einstellungen durch Schließen und erneutes Öffnen verändern.

Anmerkung: Wird IOCB #7 für eigene Routinen benutzt (und vorausgesetzt, dass er geöffnet ist), ist das Programm nicht ohne Modul lauffähig.

---

76) Kapitel IV, Abschnitt 2.2.
77) Kapitel VI, Abschnitt 7.9.
78) Kapitel IV, Abschnitt 4.2.

4.4 Verfügbarer Speicher

Möglicherweise arbeitet man an einem größeren Programm und stößt dann an die Grenzen der Speicherkapazität. Sollte das passieren, bleiben drei Möglichkeiten in Abhängigkeit davon, was man gerade tut, wenn dieses Problem auftritt.

4.4.1 Editieren

Programm sofort mit '<CTRL><SHIFT>W' abspeichern. Dann zurück zum Monitor und das System erneut booten (BOOT eingeben). Danach wieder in den Editor gehen und das Programm erneut laden.

4.4.2 Kompilieren

In den Editor gehen und das Programm abspeichern. Dann vom Monitor aus neu booten und das Programm vom Speicherlaufwerk kompilieren.

4.4.3 System ist abgestürzt

Lässt sich der Rechner nicht mehr zu normalen I/O-Operationen überreden, bleibt nur noch <RESET> als letztes Mittel übrig. In diesem Fall ist das im Speicher befindliche Programm mit hoher Wahrscheinlichkeit verloren. Daher empfiehlt es sich, zwischendurch den aktuellen Programmierstand abzuspeichern.

VI. Die Action!-Library

1 Allgemeines

Die Library stellt eine ganze Reihe an Routinen für I/O-Operationen und Grafik bereit, sodass man dafür keine Routinen zu schreiben braucht. Das Action!-Modul enthält fast 70 Routinen, die genutzt werden können. Das spart Zeit beim Programmieren.

1.1 Vokabular

Die meisten in diesem Kapitel verwendeten Begriffe wurden schon erläutert. Es bleiben noch zwei Begriffe übrig, die jetzt häufig verwendet werden - IOCB und Kanal.

IOCB steht für "Input Output Control Block" (Ein-/Ausgabe-Kontrollblock). Die CIO (Central I/O) benutzt IOCBs für die Ausführung von I/O-Funktionen. Die I/O-Routinen der Library öffnen einen IOCB, um der CIO mitzuteilen, was die Routine tun soll; dann erfolgt ein direkter Aufruf der CIO.

Die IOCBs sind nummeriert (0-7). Verwendet man Routinen, die Kanalnummern verlangen, so ist dies dann die Nummer des IOCBs, der die Informationen über das gewählte Peripheriegerät enthält. Das muss nicht heißen, dass verschiedene IOCB auch verschiedene Peripheriegeräte ansprechen. Man muss einen IOCB erst eröffnen, damit er das gewählte Peripheriegerät ansprechen kann. Das wird von der Library Routine "Open" erledigt, dürfte also nicht weiter schwerfallen.

Der Begriff "default channel" (Standardkanal) bezieht sich auf den IOCB, den Action! selbst für den Bildschirm öffnet und nutzt. Das bedeutet, dass I/O-Routinen, die den "default channel" nutzen, Informationen auf dem Bildschirm ausgeben bzw. von dort holen (E:).

Anmerkungen: Der "default channel" ist Kanal 0. Mehr zu den IOCBs enthält das Handbuch zum Betriebssystem (Operating System Reference Manual) des ATARI.

1.2 Beschreibungsformat der Library-Routinen

Die Library-Routinen werden in einem Format vorgestellt, das schnell klarmacht, wie Anwendung und Aufruf zu handhaben sind. Nachfolgend eine der Routinen als Beispiel dafür, welcher Teil des präsentierten Formats welche Information enthält. Die verwendete Routine ist "Locate".

Beispiel des Library-Formats

BYTE FUNC Locate[79]

Zweck: Die Farbe oder das Zeichen an einer bestimmten Bildschirmposition festlegen.

Format: BYTE FUNC Locate(CARD col, BYTE row)

Parameter: col eine im aktuellen Grafikmodus zulässige Spaltenzahl.
row eine im aktuellen Grafikmodus zulässige Zeilenzahl.

Beschreibung: Diese Routine liefert den ATASCII-Code eines Zeichens oder die Nummer einer Farbe an der festgelegten Bildschirmposition zurück. Die von dieser Routine

79) Kapitel VI, Abschnitt 5.8.

benutzen Register werden so hochgezählt, als ob der Cursor in der aktuellen horizontalen Bildschirmzeile weiterfahren würde. Nach der letzten Position in einer Zeile folgt dann die Erste in der nächsten Zeile! Alle Get-, Put-, Print- und Input-Routinen benutzen diese Register zum Feststellen der aktuellen Cursorposition. Daher kann man diese Routine dazu verwenden, auf eine Speicherstelle zu zeigen; eine andere Routine, um dann an dieser Stelle etwas zu verändern.

-- Ende des Beispiels --

Die Überschrift enthält den Namen der Routine einschließlich des Typs (hier BYTE FUNC). Danach folgt eine kurze Beschreibung des Zwecks, dann das Format der Routine als Deklaration. Die Deklarationsform wurde anstelle der Aufrufform gewählt, weil sie mehr über die Routine selbst mitteilt. Dazu gehören auch:

- Typ der Routine (PROC oder FUNC)
- alle Parameter
- der Datentyp jedes Parameters

Im Anschluss daran werden die von der Routine benötigten Parameter einzeln erläutert. Als Letztes folgt dann eine Beschreibung über den generellen Gebrauch der Routine sowie einiger besonderer Bedingungen.

2 Output-Routinen

Die Action!-Library enthält eine fast vollständige Sammlung an Routinen für die Ausgabe von numerischen Daten und Zeichen über jeden Kanal.

Die beiden Basisroutinen "Print" und "Put" verfügen über Optionen, mit deren Hilfe man die Ausgabe einem bestimmten Kanal zuordnen und/oder ein EOL-Zeichen (<RETURN>) nach den Daten ausgeben kann.

2.1 Die Print-Prozeduren

Alle Print-Prozeduren haben eins gemeinsam: Sie beginnen mit dem Wort "Print". Daraus ersieht man, etwas soll irgendwohin gePRINTet werden. Aber wohin? Die Antwort erhält man von den an das Wort "Print" angehängten Optionen.

Diese Optionen bestehen jeweils aus einem einzigen Buchstaben. Insgesamt sind bis zu drei verschiedene Optionen gleichzeitig zugelassen, da ja jede Option einen anderen Teil der Ausgabe kontrolliert. Und so sieht das Format von Print mit allen Optionen aus:

Print<data type>{D}{E}(<parameter>)

wobei

Print der eigentliche Funktionsname ist.

<data type> den auszugebenden Datentyp[80] mitteilt.
B (Daten des Typs BYTE)
C (Daten des Typs CARD)
I (Daten des Typs INT)

80) Kapitel IV, Abschnitt 3.3.

nichts (ein String, der einen Inhalt haben muss!)

D für "Device" (Gerät) steht. Damit wird festgelegt, welcher Kanal für die Ausgabe genommen werden soll.

E für EOL (End Of Line) steht, was einen <RETURN> ausgibt.

<Parameter> die von der Prozedur benötigten Parameter sind, die in der Anzahl unterschiedlich sein können.

Anmerkung: 'D' und 'E' sind optional. Fehlt die Angabe eines Datentyps, wird die ein String angenommen.

Dazu folgende Tabelle:

Optionen	**Strings**	**BYTEs**	**CARDs**	**INTs**
keine	Print	PrintB	PrintC	PrintI
EOL	PrintE	PrintBE	PrintCE	PrintIE
Device	PrintD	PrintBD	PrintCD	PrintID
Device & EOL	PrintDE	PrintBDE	PrintCDE	PrintIDE

Die Prozeduren wurden nach den Datentypen angeordnet, die sie ausgeben. So werden sie auch in den nachfolgenden Abschnitten behandelt.

Eine Print-Prozedur ist in obiger Liste nicht enthalten, weil es sich um einen Spezialfall handelt, was die Ausgabe angeht. Sie heißt 'PrintF' und erlaubt die Formatierung von Ausgaben, die Zahlen und Strings enthalten. Ihr ist ein eigener Abschnitt gewidmet.

2.1.1 Strings ausgeben

Es gibt vier Ausgabeprozeduren für Strings mit allen Optionen.

Zweck: Ausgabe von Strings mit Formatieroptionen

Formate: PROC Print(<string>)
PROC PrintE(<string>)
PROC PrintD(BYTE channel, <string>)
PROC PrintDE(BYTE channel, <string>)

Parameter: <string> ist entweder eine Stringkonstante oder die Kennung eines BYTE ARRAYs, das als String ausgegeben werden soll
<channel> ist eine zulässige Kanalnummer (0-7)

Beschreibung: Diese vier Prozeduren geben Strings wie folgt aus:

Print Ausgabe des Strings an den Standardkanal
PrintE wie oben mit <RETURN> am Ende
PrintD Ausgabe des Strings an den angegebenen Kanal
PrintDE wie oben mit <RETURN> am Ende

Die Anwendung an sich ist unkompliziert. Man muss nur daran denken, dass ein eventuell benötigter Kanal zuvor geöffnet werden muss.

Anmerkung: 'Print' erwartet einen String bzw. die Adresse, an der er im Speicher steht. Dazu muss der String einen Inhalt haben. 'PRINT()' ist daher nicht zulässig, da die CPU-Register A und X dann nicht mit einer String-Adresse geladen werden können, sondern zufällige Werte enthalten, was zur Ausgabe von Datenmüll führt. Bei Bedarf für die Ausgabe einer Leerzeile also 'PrintE(" "), 'PutE()' oder 'Put(155)' einfügen.

2.1.2 BYTE-Zahlen ausgeben

Mit den nächsten vier Prozeduren werden Daten des BYTE-Typs im Dezimalformat ausgegeben.

Zweck: Ausgabe eines Bytes als Dezimalzahl

Formate:
PROC PrintB(BYTE number)
PROC PrintBE(BYTE number)
PROC PrintBD(BYTE channel ,number)
PROC PrintBDE(BYTE channel ,number)

Parameter:
number ist ein arithmetischer Ausdruck, der eine Variable oder Konstante beinhalten kann
channel ist eine zulässige Kanalnummer (0-7)

Beschreibung: Ausgabe der BYTEs wie folgt

PrintB Ausgabe des Bytes an den Standardkanal[81]
PrintBE wie oben mit <RETURN> am Ende
PrintBD Ausgabe des Bytes an den angegebenen Kanal
PrintBDE wie oben mit <RETURN> am Ende

2.1.3 CARD-Zahlen ausgeben

Zweck: Ausgabe von Zahlen als CARDs im Dezimalformat

Formate:
PROC PrintC(CARD number)
PROC PrintCE(CARD number)
PROC PrintCD(CARD channel, number)
PROC PrintCDE(CARD channel, number)

Parameter:
number ist ein arithmetischer Ausdruck
channel zulässige Kanalnummer (0-7)

Beschreibung: Ausgabe von CARDs wie folgt

PrintC Ausgabe einer CARD an den Standardkanal
PrintCE wie oben mit <RETURN> am Ende
PrintCD Ausgabe einer CARD an den angegebenen Kanal
PrintCDE wie oben mit <RETURN> am Ende

81) Kapitel VI, Abschnitt 1.1.

2.1.4 INT-Zahlen ausgeben

Zweck: Ausgabe von INTs im Dezimalformat

Formate: PROC PrintI(INT number)
PROC PrintIE(INT number)
PROC PrintID(INT channel, number)
PROC PrintIDE(INT channel, number)

Parameter: number ist ein arithm. Ausdruck
channel ist eine zulässige Kanalnummer (0-7)

Beschreibung: Ausgabe von INTs wie folgt

PrintI Ausgabe von INT an Standardkanal
PrintIE wie oben mit <RETURN> am Ende
PrintID Ausgabe von INT an angegebenen Kanal
PrintIDE wie oben mit <RETURN> am Ende

2.1.5 PROC PrintF - Formatierte Ausgabe

Mit PrintF können Zahlen und Strings in der gleichen Zeile unter Zuhilfenahme eines "Format Control Strings" ausgegeben werden. Durch diesen String wird der Prozedur übermittelt, wie die Ausgabe aussehen soll.

Zweck: formatierte Ausgabe von Daten

Format: PrintF("<control string>", <data>l:, <data>:l)

Argumente: <control string> Dieser String besteht aus der Formatsteuerung und dem 'String'-Text. Der Text wird direkt ausgegeben. Die Steuersequenzen (max. 5!) enthalten Informationen für die Ausgabe der als Parameter übergebenen Daten.

<data> ist ein arithmetischer Ausdruck, der durch die Formatsteuerung formatiert wird. Die erste Steuerungsangabe bestimmt die Ausgabe des ersten <data>, die zweite Angabe das zweite <data> usw.

Beschreibung: Dies ist eine wohldurchdachte Prozedur für die Ausgabe von formatierten Daten an den Standardkanal. Bis zu fünf verschiedene Datenelemente können bei der Ausgabe in einen String eingefügt werden, jedes Element mit seinem eigenen Ausgabeformat. Die Steuercodes folgen auf der nächsten Seite.

<Code>	**formatierte Ausgabe**
%S	String
%I	INT
%U	Unsignierte CARD
%C	CHARacter

<Code>	**formatierte Ausgabe**
%H	unsignierte Hexzahl
%%	das Zeichens '%'
%E	EOL (<RETURN>)

Die Codes '%E' und '%%' manipulieren bzw. benötigen keine Datenelemente. Sie sind für die Änderung der Seitenformatierung und nicht zur Datenformatierung erforderlich.

Zeichen innerhalb des <control string>, die selber keine Steuerungsinformation darstellen, werden direkt ausgegeben, so als ob es Strings wären.

2.2 Die Put-Prozeduren

Die Put-Prozeduren werden für die Ausgabe von einzelnen Zeichen verwendet (z.B. Ausgabe von Daten des Typs BYTE als ein einzelnes ATASCII-Zeichen). Die Optionen sind ähnlich denen der Print-Prozeduren, weshalb diese nicht erneut erläutert werden.

Zweck: Ausgabe eines einzelnen ATASCII-Zeichens unter Benutzung besonderer Formatieroptionen

Formate:

```
PROC Put(CHAR[82] character)
PROC PutE()
PROC PutD(BYTE channel, CHAR character)
PROC PutDE(BYTE channel, CHAR character)
```

Parameter:

character	ein arithm. Ausdruck
channel	ist eine zulässige Kanalnummer (0-7)

Beschreibung: Ausgabe von Zeichen wie folgt

Put	Ausgabe von Zeichen an den Standardkanal
PutE	Ausgabe eines EOL (<RETURN>) an den Standardkanal
PutD	Ausgabe von Zeichen an den angegebenen Kanal
PutDE	wie PutD, zusätzlich <RETURN> am Ende

3 Input-Routinen

Es folgen nun die komplementären Routinen zu Print und Put, also die Eingaberoutinen, die Daten in das System übernehmen. Ähnlich wie bei den Ausgaberoutinen werden Datentyp und Quelle durch die Verwendung von Optionen definiert. 'Input' und 'Get' sind diese Eingaberoutinen, von denen ebenfalls jede ihre eigenen Optionen hat.

Die Input-Routinen sind in zwei Kategorien aufgeteilt: eine für die Eingabe numerischer Daten, eine für die Eingabe von Strings. Jede wird separat behandelt.

Es gibt nur eine Get-Routine (GetD), die im letzten Abschnitt erklärt wird.

3.1 Numerische Eingaben

Die sechs Funktionen für Eingaben jeder Art an numerischen Daten von einem beliebigen Kanal wurden zusammengefasst, weil sie leicht verständlich sind und anders als die numerischen Ausgaberoutinen keiner zusätzlichen Erläuterung bedürfen.

Zweck: Eingabe numerischer Daten

Formate:

```
BYTE FUNC InputB()
BYTE FUNC InputBD(BYTE channel)
```

---

82) Kapitel IV, Abschnitt 2.

CARD FUNC InputC()
CARD FUNC InputCD(BYTE channel)
INT FUNC InputI()
INT FUNC InputID(BYTE channel)

Parameter: channel ist eine zulässige Kanalnummer (0-7)

Beschreibung: Eingabe von numerischen Daten wie folgt

InputB	Eingabe einer BYTE-Zahl vom Standardkanal
InputBD	Eingabe einer BYTE-Zahl vom angegebenen Kanal
InputC	Eingabe einer CARD-Zahl vom Standardkanal
InputCD	Eingabe einer CARD-Zahl vom angegebenen Kanal
InputI	Eingabe einer INT-Zahl vom Standardkanal
InputID	Eingabe einer INT-Zahl vom angegebenen Kanal

3.2 String-Eingabe

Die Eingabe von Strings wird durch Anhängen eines "S" zum Begriff "Input" bewirkt. Es gibt drei Prozeduren für die Eingabe von Strings.

Zweck: Eingabe von String-Daten.

Formate: PROC InputS(<string>)
PROC InputSD(BYTE channel, <string>)
PROC InputMD(BYTE channel, <string>, BYTE max)

Parameter:		
	<string>	Kennung eines BYTE ARRAY
	channel	ist eine zulässige Kanalnummer (0-7)
	max	Maximal zulässige Länge für die Eingabe eines Strings. Ein zu langer String wird entsprechend abgeschnitten.

Beschreibung: Die Prozeduren erledigen die Aufgaben

InputS	Eingabe eines Strings von max. 255 Zeichen vom Standardkanal.
InputSD	Eingabe eines Strings von max. 255 Zeichen vom angegebenen Kanal.
InputMD	Eingabe eines Strings von 'max' Zeichen vom angegebenen Kanal.

Anmerkung:
Bei den String-Eingabe-Funktionen muss Platz für das EOL-Zeichen mit eingerechnet werden. Es werden also maximal 254 Zeichen plus EOL übernommen..

3.3 CHAR FUNC GetD

Zweck: Eingabe eines einzelnen Zeichens von einem angegebenen Kanal.

Format: CHAR FUNC GetD(BYTE channel)

Parameter: channel ist eine zulässige Kanalnummer (0-7)

Beschreibung: Wird verwendet, um ein einzelnes Zeichen über den angegebenen Kanal einzugeben. Das Zeichen wird von der Funktion als ATASCII-Wert zurückgegeben.

4 Routinen für die Datei-Behandlung

Dieser Abschnitt ist den Routinen gewidmet, die Peripheriegeräte ansprechen (Drucker, Diskettenstation, Cassette usw.). Mithilfe dieser Routinen werden Kanäle (IOCBs) geöffnet, geschlossen und die weitgehende Bearbeitungen von Dateien ermöglicht.

4.1 PROC Open

Zweck: Öffnet einen IOCB-Kanal für I/O-Operationen mit einem Peripheriegerät.

Format: PROC Open(BYTE channel, <filestring>, BYTE mode, aux2)

Parameter:

channel	ist eine zulässige Kanalnummer (0-7)
<filestring>	eine String-Konstante (oder die Array-Kennung dieser String-Konstante), benutzt für das Gerät (D:, P:, S: usw.), das auf dem angegebenen Kanal (IOCB #) geöffnet werden soll. Bei "D:" muss ein Dateiname mit angegeben werden.
mode	ist die Art der beabsichtigten I/O als Zahl:
4	nur lesen
6	Directory lesen
8	nur schreiben
9	anhängen
12	lesen/schreiben
aux2	abhängig vom Peripheriegerät (meist 0)

Beschreibung: Diese Prozedur öffnet einen Kanal auf das im <filestring> angegebene Gerät. Der I/O-Modus kann bestimmt werden. Die geräteabhängigen Codes werden per aux2 übergeben.

Warnung: Nicht Kanal 7 öffnen, da der von Action! für die Eingabe vom Editor genutzt wird. Für die Eingabe von der Tastatur (K:) kann man den Kanal natürlich benutzen. Dann muss aber ein offener Kanal 7 bereits vorhanden sein. Das Programm ist dann nur noch mit eingestecktem Action!-Modul lauffähig. Action! öffnet den IOCB #7 automatisch.

4.2 PROC Close

Zweck: Einen Kanal zu einem Gerät schließen.

Format: PROC Close(BYTE channel)
Parameter: channel ist eine zulässige Kanalnummer (0-7)

Beschreibung: Diese Prozedur schließt den angegebenen Kanal. Am Ende eines Programms sollten alle während des Ablaufs geöffneten IOCBs wieder geschlossen werden.

Anmerkung: Aber nicht Kanal 7 schließen, da er von Action! benötigt wird!

4.3 PROC XIO

Zweck: direkter Systemaufruf

Format: PROC XIO(BYTE channel,0,cmd,aux1,aux2,<filestring>)

Parameter:

channel	ist eine zulässige Kanalnummer (0-7)
cmd	das entsprechende IOCB COMMAND Byte
aux1	erstes Hilfsbyte im IOCB

aux2	zweites Hilfsbyte im IOCB
<filestring>	eine Zeichenkette, die ein bestimmtes Gerät anspricht.

Beschreibung: Systemaufruf für den direkten Zugriff auf das DOS; aus ATARI BASIC übernommen. Die Liste der gültigen XIO-Befehle findet sich im Handbuch des verwendeten DOS. Sie weichen teilweise sehr voneinander ab. Für NOTE und POINT stellt Action! eigene Routinen bereit. Der zweite Parameter muss immer '0' sein.

4.4 PROC Note

Zweck: Auslesen des aktuellen Dateisektors und des Offsets in den Sektor auf der angegebenen Diskettenstation.

Format: PROC Note(BYTE channel, CARD POINTER sector, BYTE POINTER offset)

Parameter:

channel	ist eine zulässige Kanalnummer (0-7)
sector	Zeiger auf die Sektornummer-Variable
offset	Zeiger auf die Byte-Offset-Variable

Beschreibung: Bei ATARI-DOS-Derivaten liefert diese Prozedur den Sektor und das Offset in den Sektor auf das nächste Byte, das geschrieben oder gelesen werden soll (Disk File Pointer).

4.5 PROC Point

Zweck: Setzen des Dateipointers (siehe oben).

Format: PROC Point(BYTE channel, CARD sector, BYTE offset)

Parameter:

channel	ist eine zulässige Kanalnummer (0-7)
sector	Sektornummer (1-max[83])
offset	siehe oben

Beschreibung: Setzen des 'Disk File Pointer' auf eine beliebige Stelle im 'Disk File'.

Anmerkung: Die Datei muss mit 'Open Append' (Modus 12) geöffnet worden sein, damit das Kommando ausgeführt werden kann.

Anmerkungen zu NOTE und POINT:
SpartaDOS und kompatible DOS (mit Ausnahme von BW-DOS) verwenden anders als die meisten DOS anstelle des absoluten Zugriffs auf den Sektor den relativen Zugriff auf die Datei. Ggf. bei der Programmierung darauf achten. Der relative Zugriff ist vorteilhafter, da er sich immer auf die Datei (Datei und Offset in die Datei) und nicht auf eine Diskettenposition (Sektor und Offset in den Sektor) bezieht. Details dazu finden sich in den entsprechenden DOS-Handbüchern. Die maximale Sektornummer hängt von DOS und Laufwerk ab.

Je nach Größe des Speichermediums und/oder der gesuchten Daten kann der relative Zugriff den Programmablauf um ein Vielfaches beschleunigen. In dem Falle ist eine eigene Routine vorteilhafter als die in der Library angebotene.

---

83) Die maximale Sektornummer hängt vom DOS und dem darunter verwendeten Format ab.

5 Grafik und Spielsteuerung

Die Action!-Library enthält einige Routinen, die speziell dafür entwickelt wurden, das Schreiben von Spielen leichter und einfacher zu gestalten. Damit ist es ein Leichtes, Bit-Map-Grafiken zu verändern, die unzähligen Soundeffekte des ATARIs auszunutzen oder Informationen von Joystick, Paddle oder anderen Spielsteuerungen einzulesen.

Da die Erläuterung der Routinen ihre Verwendung am besten verdeutlicht, geht es direkt ohne weitere Erklärungen mit den Routinen weiter.

5.1 PROC Graphics

Zweck: Bit-Map-Grafik des ATARIs einschalten.

Format: PROC Graphics(BYTE mode)

Parameter: mode Nummer des Grafikmodus[84] wie in ATARI BASIC

Beschreibung: Diese Routine entspricht exakt der des Graphics-Befehls in ATARI-BASIC. Detaillierte Erläuterungen zu den Grafikmodi enthält das ATARI-Profibuch vom ABBUC.

Anmerkung: Alle Grafikmodi zeigen einen geteilten Bildschirm. Wie in BASIC addiert man zum 'mode' 16 für das Vollbild
32 zum Erhalten des Bildinhalts und
48 für beides.

5.2 PROC SetColor

Zweck: Setzt in den angegebenen Farbregistern die Werte.

Format: PROC SetColor(BYTE register,hue,luminance)

Parameter: register eines der fünf Farbregister (0-4)
hue Farbe
luminance Helligkeit

Beschreibung: Die Routine entspricht dem Befehl SetColor in ATARI-BASIC.

Anmerkung: Es gibt 16 Farben (0-15) und 8 Helligkeitswerte (0-14 in 2er-Schritten), woraus die 128 Farben des ATARI gemischt werden.

Die Grundeinstellung der Farbregister des ATARIs :

Register	Farbe	Helligkeit
0	2	8
1	12	10
2	9	4
3	4	6
4	0	0

84) ABBUC-Edition des ATARI Profibuchs.

Die Farbtöne bei NTSC- und PAL-Geräten weichen voneinander ab. Die Farbtabelle für NTSC-Geräte enthält das ATARI-Profibuch vom ABBUC. Für PAL-Geräte gibt es keine genormte Farbtabelle. Außerdem hängen die Farbtöne auch stark vom verwendeten ATARI-Modell und dessen Einstellung sowie vom Monitor und dessen Einstellungen ab.

5.3 BYTE color

'color' ist keine Library-Routine, sondern eine definierte Variable für die Verwendung mit 'Plot', 'DrawTo' und 'Fill'. Grafikmodus einschalten (mit 'Graphics'), Farbregister setzen (mit 'SetColor') und schon kann man in dem Grafikmodus 'Plot' und 'DrawTo' mit den Farben verwenden, die zuvor gesetzt wurden mit

color=<number>

wobei

<number> das Farbregister angibt, in dem die Farbe gespeichert ist, die verwendet werden soll.

Anmerkung: Erläuterungen zum Zusammenhang zwischen Farbregistern und Grafikmodi finden sich im ATARI-Profibuch vom ABBUC.

5.4 PROC Plot

Zweck: Cursor auf eine bestimmte Position setzen und dann die mit der Variablen 'Color' festgelegte Farbe verwenden.

Format: PROC Plot(CARD col,BYTE row)

Parameter:
- col horizontale Koordinate
- row vertikale Koordinate

Beschreibung: Wird in Grafikstufe 3 - 8 verwendet, um einen Punkt auf den Bildschirm zu 'plot'ten. Die Größe des Punkts hängt ab vom Grafikmodus, die Farbe wird durch den Wert in der Library-Variablen 'Color' bestimmt (siehe 5.3).

5.5 PROC DrawTo

Zweck: Zeichnen einer Linie von einem zuvor ge'plot'tetem Punkt zur aktuellen Cursorposition.

Format: PROC DrawTo(CARD col,BYTE row)

Parameter:
- col horizontale Koordinate des Endpunkts der Linie
- row vertikale Koordinate des Endpunkts der Linie

Beschreibung: Wird in Grafikstufe 3 - 8 verwendet, um eine Linie von einem gerade ge'plot'tetem Punkt zu dem durch die Koordinaten bestimmten Endpunkt zu zeichnen. Die Farbe wird durch den Wert der Library-Variablen 'Color' bestimmt (siehe 5.3).

5.6 PROC Fill

Zweck: Füllen einer Fläche mit Farbe.

Format: PROC Fill(CARD col,BYTE row)

Parameter: col horizontale Koordinate der unteren rechten Ecke der zu füllenden Fläche
row vertikale Koordinate der unteren rechten Ecke der zu füllenden Fläche

Beschreibung: Damit lassen sich in den Grafikmodi 3 - -8 gefüllte Rechtecke erzeugen. Die obere linke Ecke des zu zeichnenden Rechtecks muss unmittelbar vor dieser PROC durch Plot (siehe 5.4) festgelegt werden. Die rechte untere Ecke wird dann durch die Parameter angegeben. Die Farbe wird durch den Wert in der Library-Variablen 'Color' bestimmt (siehe 5.3).

5.7 PROC Position

Zweck: Den Cursor irgendwo auf dem Bildschirm positionieren.

Format: PROC Position(CARD col,BYTE row)

Parameter: col horizontale Koordinate
row vertikale Koordinate

Beschreibung: Setzt den Cursor an die definierte Position, und zwar in jedem Grafikmodus. Print, Put, Input und Get nutzen diese Positionsregister.

5.8 BYTE FUNC Locate

Zweck: Die Farbe oder das Zeichen an einer bestimmten Bildschirmposition festlegen.

Format: BYTE FUNC Locate(CARD col, BYTE row)

Parameter: col eine im aktuellen Grafikmodus zulässige Spaltenzahl.
row eine im aktuellen Grafikmodus zulässige Zeilenzahl.

Beschreibung: Diese Routine liefert den ATASCII-Code eines Zeichens oder die Nummer einer Farbe an der festgelegten Bildschirmposition zurück. Die von dieser Routine benutzen Register werden so hochgezählt, als ob der Cursor in der aktuellen horizontalen Bildschirmzeile weiterfahren würde (nach der letzten Position in einer Zeile folgt dann die Erste in der nächsten Zeile!). Alle Get-, Put-, Print- und Input-Routinen benutzen diese Register zum Feststellen der aktuellen Cursorposition. Daher kann man diese Routine dazu verwenden, auf eine Speicherstelle zu zeigen; eine andere Routine, um dann an dieser Stelle etwas zu verändern.

5.9 PROC Sound

Zweck: Soundfähigkeiten des ATARI nutzen.

Format: PROC Sound(BYTE voice,pitch,distortion,volume)

Parameter: voice einer der 4 Tongeneratoren (0-3)
pitch Tonhöhe
distortion Verzerrung
volume Lautstärke

Beschreibung: Diese PROC ermöglicht die Steuerung der Tongeneratoren des ATARIs, so wie in BASIC. Verzerrung 10 ist der einzige Wert, der sich für Musik eignet. Alle anderen Verzerrungswerte sind hilfreich für andere Geräuscheffekte wie Flugzeug, Rennwagen, etc.

Anmerkung: Detaillierte Erläuterungen zu den Fähigkeiten der Tongeneratoren sowie der Tonhöhen finden sich im ATARI-Profibuch vom ABBUC.

5.10 PROC SndRst

Zweck: Alle Soundregister auf null setzen.

Format: PROC SndRst()

Parameter: keine

Beschreibung: Setzt alle Soundregister auf null, sodass kein Ton mehr erzeugt wird.

5.11 BYTE FUNC Paddle

Zweck: Einlesen des aktuellen numerischen Paddle-Wertes.

Format: BYTE FUNC Paddle(BYTE Port)

Parameter: port Portnummer des verwendeten Paddles
(0-7 bei 400/800, 0-3 bei XL/XE)

Beschreibung: Liest den aktuellen Wert des angesprochenen Paddles aus.

5.12 BYTE FUNC PTrig

Zweck: Feststellen, ob ein Paddle-Trigger gedrückt wurde.

Format: BYTE FUNC Paddle(BYTE port)

Parameter: port Portnummer des verwendeten Paddles
(0-7 bei 400/800, 0-3 bei XL/XE)

Beschreibung: Diese Funktion liefert den aktuellen Trigger-Wert eines angesprochenen Paddles zurück. Ist der Trigger gedrückt, wird der Wert 0 zurückgegeben, andernfalls ist der Wert verschieden von 0.

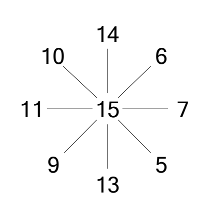
5.13 BYTE FUNC Stick

Zweck: Gibt den aktuellen numerischen Wert des angesprochenen Joysticks zurück.

Format: BYTE FUNC Stick(BYTE port)

Parameter: port siehe 5.11

Beschreibung: Diese Funktion liefert die momentane Position des Joysticks entsprechend der Codierung des Diagramms.



5.14 BYTE FUNC STrig

Zweck: Feststellen, ob ein Joystick-Trigger gedrückt wurde.

Format: BYTE FUNC STrig(BYTE port)

Parameter: port siehe 5.11

Beschreibung: Diese Funktion liefert den momentanen Wert des angesprochenen Joystick-Triggers. Ist der Trigger gedrückt, wird der Wert 0 ausgegeben, andernfalls ein Wert verschieden von 0.

6 Behandlung / Umwandlung von Strings

Mit den nachfolgend erläuterten Routinen lassen sich Strings manipulieren, eine Zahl in einen String und ein String in eine Zahl umwandeln. Die Routinen sprechen für sich selbst.

6.1 Routinen zur String-Behandlung

Die nächsten vier Routinen erlauben fortgeschrittene String-Manipulationen; und zwar Vergleich und Kopie eines Strings sowie die Einfügung eines Teilstrings. Eines dabei bitte nicht vergessen: Die maximale Länge eines Strings beträgt 255 Zeichen. Deshalb niemals versuchen, diese Funktionen für die Einrichtung oder Manipulation großer CHARacter-Arrays zu verwenden.

6.1.1 INT FUNC SCompare

Zweck: zwei Strings alphabetisch vergleichen.

Format: INT FUNC SCompare(<string1>,<string2>)

Parameter: <string1> ist ein String mit Anführungszeichen oder die Kennung eines CHAR ARRAY, die ein String ist.
<string2> s.o.

Beschreibung: Diese Funktion liefert einen Wert nach folgender Tabelle

Vergleich	**liefert Wert**
<string1> < <string2>	Wert < 0
<string1> = <string2>	Wert = 0
<string1> > <string2>	Wert > 0

Der Vergleich erfolgt alphabetisch, daher lassen sich Listen aus Strings damit hervorragend sortieren.

6.1.2 PROC SCopy

Zweck: Einen String in einen anderen kopieren.
Format: PROC SCopy(<dest>,<source>)

Parameter: <dest> ist die Kennung des Strings (CHAR ARRAY) für die Kopie.
<source> ist ein String in Anführungszeichen oder die Kennung des CHAR ARRAY, also die Vorlage für die Kopie.

Beschreibung: Diese Prozedur kopiert den Inhalt von <source> nach <dest>; dabei wird <dest> durch <source> ersetzt.

Anmerkung: Es scheint von der Art der Deklaration abzuhängen, wie diese Routine arbeitet. Ist <source> größer als <dest>, können Fehler auftreten.

6.1.3 PROC SCopyS

Zweck: Einen Teil eines Strings in einen anderen String kopieren.

Format: PROC SCopyS(<dest>,<source>,BYTE start,stop)

Parameter:
- <dest> ist die Kennung des Strings (CHAR ARRAY) für die Kopie.
- <source> ist ein String in Anführungszeichen oder die Kennung des CHAR ARRAY, also die Vorlage für die Kopie.
- start ist der Startpunkt für die Kopie in <source>.
- stop ist der Endpunkt für die Kopie in <source>. Sollte 'stop' größer sein als die Länge von <source>, wird 'stop' auf die Länge von <source> geändert.

Beschreibung: Diese Prozedur kopiert aus <source> den Ausschnitt von 'start' bis 'stop' nach <dest>; <dest> wird durch den Ausschnitt aus <source> ersetzt.

6.1.4 PROC SAssign

Zweck: Kopiert einen String in den Teil eines anderen Strings.

Format: PROC SAssign(<dest>,<source>,BYTE start,stop)

Parameter:
- <dest> ist die Kennung des Strings (CHAR ARRAY) für die Kopie.
- <source> ist ein String in Anführungszeichen oder die Kennung des CHAR ARRAY, also die Vorlage für die Kopie.
- start ist der Startpunkt für das Hineinschreiben der Kopie in <dest>.
- stop ist der Endpunkt für das Hineinschreiben der Kopie in <dest>. Ist die Position von 'stop' größer als die Länge von <source>, wird 'stop' auf die Länge von <source> herabgesetzt.

Beschreibung: Diese Prozedur kopiert einen String (<source>) in einen anderen String (<dest>) als Teil hinein. Die Kopie wird in <dest> von der Position 'start' bis 'stop' hineingeschrieben. Sollte der angegebene Platzbedarf (stop-start+1) größer sein als die Länge von <source>, dann wird 'stop' entsprechend herabgesetzt. So wird vermieden, dass über das Ende von <source> hinaus fehlerhafte Daten in den String <dest> hineinkopiert werden.

Anmerkungen: Diese Art des Kopierens überschreibt in <dest> die alten Elemente mit denen aus <source>. Es wird also nicht eingefügt!

6.2 Konvertieren von Zahlen in Strings

Die nächsten drei Prozeduren konvertieren eine als Parameter angegebene Zahl in eine Zeichenkette. Für jeden numerischen Datentyp gibt es eine eigene Prozedur.

Zweck: Umwandeln einer Zahl in einen String.

Format: PROC StrB(BYTE number,<string>)
PROC StrC(CARD number,<string>)
PROC StrI(INT number,<string>)

Parameter: number ist ein arithmetischer Ausdruck (nicht vergessen, dass ein arith metischer Ausdruck nur eine Konstante oder ein Variablen name sein darf).
<string> ist die Kennung eines CHAR ARRAYs.

Beschreibung: Diese Prozedur verwandelt BYTE-, CARD- und INT-Werte in Zeichenketten, die aus den Ziffern der vorgegebenen Zahlen zusammengesetzt werden. Beispiel: Aus der Zahl mit dem Wert 35 wird der String "35".

6.3 Konvertieren von Strings in Zahlen

Zweck: Konvertieren einer aus Ziffern bestehenden Zeichenkette in eine Zahl.

Format: BYTE FUNC ValB(<string>)
CARD FUNC ValC(<string>)
INT FUNC ValI(<string>)

Parameter: <string> ist ein String in Anführungszeichen oder dieKennung eines CHAR ARRAYs, bestehend aus den Ziffern "0"-"9".

Beschreibung: Diese Routine liefert in Abhängigkeit von der Funktion den numerischen Wert (BYTE, CARD oder INT) eines vorliegenden Strings.

7 Sonstige Routinen

Dieser Abschnitt behandelt diejenigen Routinen, die in keine der bisherigen Kategorien hineinpassen, die aber trotzdem nützlich ist.

Es handelt sich um folgende Routinen:

Rand	Zufallszahlengenerator
Break	nützlich zum Entfehlern
Error	eine Systemroutine, die man ersetzen kann
Peek	eine Speicherstelle auslesen (BYTE)
PeekC	zwei Speicherstellen auslesen (als CARD)
Poke	einen BYTE-Wert in eine Speicherstelle schreiben
PokeC	einen CARD-Wert in den Speicher schreiben
Zero	einen Speicherbereich löschen (ausnullen)
SetBlock	einen Speicherbereich mit einem Wert füllen
MoveBlock	einen Speicherbereich verschieben
Device	die Variable für den I/O-Kanal
Trace	steuert die Trace-Option des Compilers
List	steuert die List-Option des Compilers
EOF	enthält den EOF-Status aller Kanäle

Diese Routinen decken ziemlich unterschiedliche Anforderungen ab, daher sind sie in diesem extra Abschnitt zusammengefasst.

7.1 BYTE FUNC Rand

Zweck: Eine Zufallszahl erzeugen.

Format: BYTE FUNC Rand(BYTE range)

Parameter: range ist der größte Wert für die Zufallszahl.

Beschreibung: Diese Funktion liefert eine Zufallszahl zwischen 0 und ('range'-1). Ist 'range'=0 wird eine Zufallszahl zwischen 0 und 255 generiert.

7.2 PROC Break

Zweck: Die Programmausführung unterbrechen.

Format: PROC Break()

Parameter: keine

Beschreibung: Mithilfe dieser Prozedur kann man die Ausführung des Programms unterbrechen, um Variablen zu überprüfen oder allgemein Fehlersuche zu betreiben. Das Programm kann dann mit dem auf die Break-Routine folgenden Anweisung fortgesetzt werden, indem man das Monitorkommando 'PROCEED' verwendet.

7.3 PROC Error

Diese Routine wird vom Action!-System selbst aufgerufen, wenn es Fehler entdeckt oder innerhalb der CIO Fehler auftreten. Man kann aber auch eine eigene Fehlerroutine schreiben und Action! diese anstelle der PROC Error benutzen lassen. Dazu müssen nur folgende Anweisungen in das eigene Programm eingebaut werden:

```
PROC MyError(BYTE errcode)

;*** Dies ist eine eigene Fehlerroutine und der Fehlercode
;*** wird an sie vom Action!-System uebergeben.
;*** Die eigenen Fehlerbehandlungsroutinen hier einfuegen.

RETURN ; End der PROC MyError


PROC main()      ; die eigene Hauptprozedur
  CARD temperr  ; nimmt die Adresse der systemeigenen
                ; Fehlerroutine (PROC Error) auf
  temperr=Error ; die Adresse der Fehlerroutine des Systems
                ; speichern
  Error=MyError ; die Adresse der systemeigenen Fehlerroutine
                ; auf den Start der eigenen Fehlerroutine
                ; zeigen lassen

               ; hier den Kern des eigenen Programms einfügen
  Error=temperr ; die Adresse der systemeigenen Fehlerroutine
                ; zuruecksetzen auf die wahre Systemroutine

RETURN          ; Ende des Programms
```

Tatsächlich passiert nichts weiter, als das der Zeiger der Fehlerroutine des Systems auf die eigene Fehlerroutine gesetzt wird. Man braucht die Fehlerroutine aber nicht selbst aufzurufen, da sie beim Auftreten eines Fehlers von Action! aufgerufen wird.

Man beachte, dass der ursprüngliche Zeiger der Fehlerroutine zwischengespeichert und am Ende des Programms auch wieder zurückgesetzt wurde. Damit kann das System nach Beendigung des Programms wieder die systemeigene Fehlerroutine benutzen.

Warnung: Vorsicht bei Experimenten mit eigenen Fehlerroutinen. Tritt ein noch nicht von der eigenen Routine erfasster Fehler auf, kann es zum Systemabsturz kommen.

7.4 BYTE FUNC Peek und CARD FUNC Peek

Zweck: Wert aus einer Speicherstelle auslesen (BYTE oder CARD)

Format: BYTE FUNC Peek(CARD address)
CARD FUNC Peek(CARD address)

Parameter: address ist die Adresse der Speicherstelle, die ausgelesen werden soll

Beschreibung: Mit diesen beiden Funktionen kann man den Inhalt einer Speicherstelle während des Programmablaufs einsehen. Und zwar entweder als BYTE oder als CARD im LSB-MSB-Format.

7.5 PROC Poke und PROC PokeC

Zweck: neue Werte (BYTE oder CARD) in eine bestimmte Speicherstelle schreiben.
Format: PROC Poke(CARD address, BYTE value)
PROC PokeC(CARD address, CARD value)

Parameter: address ist die Adresse einer Speicherstelle.
value ist der Wert, der dort hineingeschrieben werden soll. Im Falle von PokeC wird der CARD-Wert in 'address' und 'address+1' im LSB-MSB-Format abgelegt.

Beschreibung: Mit diesen PROCs kann man während des Programmablaufs den Inhalt von Speicherstellen verändern.

7.6 PROC Zero

Zweck: Einen Speicherblock löschen.

Format: PROC Zero(BYTE POINTER address,CARD size)

Parameter: address ist der Zeiger auf die Startadresse des Blocks, der gelöscht werden soll.
size ist die Größe des zu löschenden Blocks.

Beschreibung: Mit dieser Prozedur kann man alle Speicherstellen des angegebenen Blocks auf den Wert 0 setzen. Der Block beginnt bei 'address' und endet bei 'address'+'size'-1.

7.7 PROC SetBlock

Zweck: Den Inhalt der Speicherstellen eines bestimmten Speicherblocks auf einen festgelegten Wert setzen.

Format: SetBlock(BYTE POINTER address, CARD size, BYTE value)

Parameter:
address ist der Zeiger auf die Startadresse des Blocks.
size ist die Größe des zu setzenden Blocks.
value ist der Wert, auf den die Bytes im Block gesetzt werden sollen.

Beschreibung:Mit dieser Prozedur kann man alle Werte in einem Speicherbereich auf 'value' setzen. Der Bereich beginnt bei 'address' und endet bei 'address' +'size'-1.

7.8 PROC MoveBlock

Zweck: Den Inhalt eines Speicherbereichs verschieben.

Format: PROC MoveBlock(BYTE POINTER dest,source, CARD size)

Parameter:
dest ist ein Zeiger auf den Anfang des Zielblocks.
source ist ein Zeiger auf den Anfang Ursprungsblocks.
Size ist die Größe des zu verschiebenden Blocks.

Beschreibung: Diese Prozedur verschiebt die Speicherinhalte eines Blocks, der bei der Adresse 'source' beginnt und bei der Adresse 'source'+'size'-1 endet, in einen anderen Block, der bei der Adresse 'dest' beginnt und bei der Adresse 'dest'+'size'-1 endet. Sollte 'dest' größer als 'source' sein und sich dazwischen nicht genügend freier Speicherplatz ('size') befinden, funktioniert diese Routine nicht richtig. Das liegt daran, dass dann ein Teil des zu verschiebenden 'source' bereits im Bereich 'dest' liegt.

7.9 BYTE Device

'device' ist eine in der Action!-Library definierte Variable, die die Steuerung des für die I/O-Funktionen festgelegten Geräts (device) ermöglicht. Die in 'device' enthaltene Zahl ist die Kanalnummer des festgelegten Geräts. Eine Ausgabe auf den Drucker wird so erreicht:

```
Close(5)
; avoid a 'File already Opened' error
Open(5,"P:",8)
device=5
```

Und kann zum Beispiel auf den Bildschirm umgelenkt werden:

```
Close(5)
; close "P:"
device=0
```

7.10 BYTE TRACE

Diese Variable aus der Library ermöglicht die Steuerung der Compiler-Option 'TRACE' aus einem Programm heraus. Man muss sie mit der Compiler-Direktive 'SET' benutzen und ganz am Anfang des Programms setzen. Setzt man 'TRACE' auf 0, wird die Option abgeschaltet, mit 1 dagegen eingeschaltet.

Beispiel: `SET TRACE=0`

7.11 BYTE LIST

Diese Variable aus der Library steuert die Compiler-Option 'LIST'. Wie 'TRACE' muss sie mit der 'SET'-Direktive ganz am Anfang des Programms gesetzt werden. 0 schaltet 'LIST' aus, 1 dagegen ein.

7.12 BYTE ARRAY EOF(8)

Mit dieser Variablen aus der Library kann man herausfinden, ob auf einem beliebigen Kanal das Ende einer Datei (EOF) erreicht wurde. Man gibt nur die Kanalnummer als Ergänzung zum EOF ARRAY an. Soll zum Beispiel herausgefunden werden, ob auf Kanal 1 das Ende der Datei erreicht ist (EOF), nutzt man:

```
IF EOF(1) THEN
     :
     :
```

EOF wird auf 1 gesetzt, wenn das Ende der Datei erreicht wurde, andernfalls auf 0.

VII. Das Action! Run Time Package

1 Einführung
Das Action! Run Time Package (ab jetzt einfach "RunTime" genannt) soll Anwender der Modul-basierten Programmiersprache Action! unterstützen. Mithilfe der RunTime kann ein Action!-Programm so kompiliert werden, dass es ohne ACTION-Modul läuft.

Der Hauptvorteil der RunTime ist die so ermöglichte Weitergabe eigener Programme an Freunde, Mitglieder der User Group, etc. Der Verkauf von in Action! geschriebenen Programmen erfordert allerdings eine kommerzielle Lizenz!

Ein weiterer Vorteil der RunTime ist die Erstellung externer Befehle (also Programme mit der Erweiterung ".COM") für die Anwendung unter OS/A+ oder DOS XL. Die neuen Befehle können jederzeit aufgerufen werden und nicht nur dann, wenn das Action!-Modul aktiviert ist.

In Abschnitt 2 wird erläutert, wie Action! Programme kompiliert und wie es seine Symboltabellen aufbaut. Daneben finden sich weitere Informationen, die beim Kompilieren von in Action! geschriebenen Programmen in nützlich sein können. Es empfiehlt sich, diesen Abschnitt sorgfältig durchzulesen.

Danach werden vier mögliche Wege für die Verwendung der RunTime aufgezeigt. Es bietet sich an, zunächst ein kurzes Programm mit der in Abschnitt 3 dargestellten Methode zu schreiben und zu kompilieren. Danach kann man die ersten Absätze der Abschnitte 4, 5 und 6 lesen, um zu sehen, ob die in diesen Abschnitten beschriebenen Methoden hilfreich sind.

Ergänzend dazu enthält der Anhang C eine Übersicht der nützlichsten und interessantesten Speicherbereiche, die entweder vom Compiler oder der Runtime verwendet werden. Viele dieser Speicherstellen werden ausführlich in anderen Abschnitten erläutert, sodass notwendige Querverweise gegeben werden.

2 Wie Action! arbeitet

2.1 Kompilieren eines Programms
Wenn der ACTON!-Monitor eine Anforderung zur Kompilierung erhält, initialisiert er vordefinierte Tabellen, setzt und verwendet bestimmte Memory Pointer und beginnt dann den 6502-Maschinencode direkt im Speicher zu erzeugen; dabei achtet er auf bestimmte Systemvariablen, die nachfolgend beschrieben werden.

In den folgenden Erläuterungen werden eckige Klammern verwendet, um Adressen zu kennzeichnen, auf die durch die benannte oder angesprochene Speicherstelle verwiesen wird. [$02E7] bedeutet also "die Speicherstelle(n), auf die durch den Inhalt der Adresse $02E7 gezeigt wird". Im Allgemeinen sind Wörter in Großbuchstaben Labels aus der Speicherübersicht in Abschnitt 7.

2.1.1 Speicheraufteilung
Ohne Änderung durch den Anwender teilt Action! den Speicher wie folgt auf:

Der Editorpuffer beginnt bei [APPMHI]. Dieser Zeiger wird selbst als Offset (von ca. $700 Bytes) abgeleitet von [LOMEM]. Der Bereich zwischen [LOMEM] und der ursprünglichen Adresse von [APPMHI] wird für verschiedene, teilweise vordefinierte Puffer, Tabellen, usw. verwendet.

Mit dem Schreiben eines Action!-Programms ändert sich [APPMHI] entsprechend.

Beim Kompilieren des Programms wird APPMHI nach CODEBASE ($0491,$0492) kopiert. Außerdem wird CODESIZE ($0493,$0494) auf null gesetzt.

Der Bereich für die Symboltabelle wird von MEMTOP ab zugewiesen. Die Symboltabelle enthält Symbole sowohl für globale als auch lokale Variablen. Welcher Teil der Tabelle wofür benutzt wird, steuern die Hash-Tabellen, die zu den bereits erwähnten vordefinierten Tabellen gehören. Die Größe des zugewiesenen Speichers wird in STSP ($0495,$0496) festgelegt, was aber durch den Anwender geändert werden kann (siehe Abschnitt 4).

Sobald der Code kompiliert ist, ändert Action! [APPMHI], um damit den Beginn des kompilierten Codes im Speicher anzuzeigen. Zusätzlich wird CODESIZE erhöht, was die Größe des erzeugten Codes anzeigt.

Nachdem der Code kompiliert wurde, greift das W-Kommando des Monitors auf CODEBASE und CODESIZE zu, um festzustellen, aus welchem Speicherbereich die Inhalte in das Objekt-File geschrieben werden sollen.

Die wichtigste Konsequenz daraus: Befindet sich KEIN Programm im Speicher, wird der Code an der niedrigstmöglichen Speicheradresse generiert. Annahme: Wenn der Code an der niedrigsten Adresse oder in Abhängigkeit von der Obergrenze des Editorpuffers an eine höhere Adresse kompiliert werden kann, kann er vielleicht an einer beliebigen Adresse kompiliert werden. Tatsächlich bewahrheitet sich diese Annahme größtenteils.

Die einzigen Limitierungen sind die vordefinierten Puffer, das kompilierte Programm und die Symboltabelle, die auf jeden Fall in den Speicher zwischen der Obergrenze des DOS ([LOMEM]) und der unteren Grenze des Bildschirmspeichers ([MEMTOP]) passen muss.

DOS XL ab Version 2.3 (DOSXL.SUP auf der Disk) bietet einen Wert für LOMEM, der mehr freien Speicher bereitstellt. Ein darunter kompiliertes Programm ist vermutlich nicht ohne Action!-Modul lauffähig, da es den unteren Speicherbereich anderer DOS überlagert. Abschnitt 5 zeigt das Kompilieren mit einem Offset und enthält weitere Informationen zu diesem Aspekt.

2.1.2 Symboltabelle durchsuchen

Sobald der Action!-Compiler auf ein Symbol trifft (z. B. den Namen einer Variablen, einer Festlegung durch DEFINE, den Namen einer TYPE oder PROC), sucht er immer an drei Stellen nach dem Symbol.

Zuerst wird die lokale Symboltabelle durchsucht. Alle Symbole, die nach den Schlüsselbegriffen PROC oder FUNC (mit Ausnahme der Namen der PROC oder FUNC) angetroffen werden, werden als lokal betrachtet. Das schließt auch die Parameter zu einer PROC oder FUNC mit ein.

Als Zweites wird die globale Symboltabelle überprüft. Die Namen aller PROCs und FUNCs befinden sich in der globalen Tabelle. Ebenso alle Namen, die vor der ersten PROC oder FUNC gefunden werden und alle Namen, die zwischen dem Schlüsselbegriff MODULE und der nächsten, nachfolgenden PROC oder FUNC stehen. Falls nicht klar

sein sollte, ob ein Name global oder lokal vergeben ist, bietet sich ein Blick in das entsprechende Kapitel an[85].

Falls der Name weder in der lokalen noch der globalen Symboltabelle gefunden wird, vermutet der Compiler, dass es sich um den Namen einer Routine aus der Action!-Library handelt. Die im Modul befindliche Library wird nach dem Namen durchsucht. Nur wenn der Name hier nicht gefunden wird, gibt Action! den Fehler "undefined symbol" (nicht definiertes Symbol) aus.

2.1.3 Symboltabelle einrichten

Beim ersten Booten des Action!-Moduls werden verschiedene Tabellen und Puffer eingerichtet (die als teilweise vordefiniert bezeichnet wurden). Diese teildefinierten Bereiche beginnen bei [LOMEM] und belegen ungefähr $700 Bytes. Von diesen $700 Bytes werden $400 Bytes für zwei je 512 Bytes große Hash-Tabellen verwendet – eine nimmt bis zu 255 lokale Symbolzeiger auf. Es gibt also eine für lokale und eine für globale Symbolzeiger. Die Suche nach und Speicherung von Symbolen in Action! erfolgt mithilfe von Hash-Algorhythmen, was solche Suchen merklich beschleunigt, aber diese extra Hash-Zeiger-Tabellen erfordert. "Hashing" ist lediglich die Anwendung einer mathematischen Formel auf ein Symbol zwecks Erzeugung eines Index – also eines Hash-Zeigers – in eine extra dafür ausgelegte Tabelle.

Wenn Action! eine Kompilierung startet, reserviert es zuerst Speicher für die Symboltabellen und die dazu gehörenden Zeiger. Es stellt per Speicherstelle STSP fest, wie viele Pages zu je 256 Bytes für die Symboltabelle benötigt werden, und weist diese dann grob ab der Obergrenze des freien Speichers (z.B. geradewegs unterhalb des Bildschirmspeichers) her nach unten zu.

Trotz der zwei Hash-Tabellen gibt es nur eine einzige Symboltabelle. Das ist aus zweierlei Gründen möglich. Erstens wird niemals nach einem Symbol unmittelbar in der Symboltabelle gesucht (da Action! immer per Hash-Tabellen-Zeiger sucht), weshalb globale und lokale Symbole in einer Tabelle zusammengefasst werden können, ohne dass es zu falschen Ergebnissen führt.

Zweitens ergänzt Action! niemals beide Tabellen zur gleichen Zeit. Action! beginnt mit der Verarbeitung der globalen Namen, dem Hinzufügen aller Variablen etc., die es in der globalen Hash-Tabelle findet und erhöht folglich die Größe der Symboltabelle. Allerdings, wenn Action! den Namen einer FUNC oder PROC kompiliert, schaltet es automatisch den Modus um – nun werden alle neuen Namen der lokalen Hash-Tabelle hinzugefügt und, als Konsequenz daraus, an das Ende der Symboltabelle angehängt. Wird als nachfolgendes Schlüsselwort MODULE, PROC oder FUNC erkannt, löscht Action! die lokale Hash-Tabelle und gibt den durch sie belegten Platz in der Symboltabelle zur Wiederverwendung frei. Da sich die lokalen Namen in der Symboltabelle immer hinter den globalen Namen befinden, stellt dieses Verfahren die größtmögliche Ausnutzung des Speicherbereichs für die Symboltabelle sicher.

Eine letzte Anmerkung dazu: Diese Vorgehensweise erklärt auch, warum der Monitor nur auf die lokalen Namen der zuletzt kompilierten PROC oder FUNC zugreifen kann (und natürlich auf alle globalen Namen).

---

85) Siehe Kapitel IV, Abschnitt 6.3.

2.2 Aufrufen eines Action!-Programms

Da der Compiler puren Maschinencode erzeugt, ist das Ausführen eines Action!-Programms unter jedem A8-DOS relativ einfach. Es muss dazu nur der übliche Loader (wie der in DOS XL, OS/A+ oder ATARI-DOS) aufgerufen und sichergestellt werden, dass die Run-Adresse des kompilierten Programms korrekt übergeben wird.

Die Run-Adresse lässt sich nach dem Kompilieren des Programms mit dem Monitor-Kommando '?' ermitteln. Angenommen der Name der letzten PROC im Programm lautet MAINPROC, dann wird mit '? MAINPROC' im Monitor die Adresse von MAINPROC angezeigt[86].

Außerdem wird mit dem W-Kommando des Monitors nicht nur der kompilierte Code ausgegeben, sondern auch ein korrekt angelegter INIT-Vektor. Details zu diesem Vektor finden sich im Handbuch des verwendeten DOS. Daher braucht man sich normalerweise keine Gedanken zu der Startadresse zu machen[87].

Allerdings ist die Art und Weise wie ein Programm geladen und gestartet wird nicht bei jedem DOS gleich. Das zu erläutern würde hier zu weit führen. Es sei angemerkt, dass ein kompiliertes Programm mit dem Extender '.COM' eine gültige Befehlsdatei für DOS XL und OS/A+ ergibt, die dann von der Eingabeaufforderung 'D1:' nur mit Eingabe des Namens (ohne Extender) aufgerufen werden kann.

2.3 Programmausführung

Unabhängig davon, ob ein Programm unter Verwendung dieser RunTime kompiliert wird oder nicht, muss es Zugriff auf eine Vielzahl an Library-Routinen nehmen können, von denen einige bereits bekannt sind[88]. Andere bleiben jedoch völlig unsichtbar. Damit ein brauchbarer Kompromiss zwischen der Größe des erzeugten Codes und dessen Arbeitsgeschwindigkeit erreicht wird, kompiliert Action! in ein Programm automatisch zahlreiche Aufrufe (JSRs) verschiedener Hilfsroutinen hinein. Beispiele dafür sind Routinen zur Multiplikation, Division und Bit-Verschiebung.

Wird ein Programm mit RunTime kompiliert, werden diese Routinen aus der Library-Bank des Moduls bereitgestellt und tatsächlich mit in das Programm übernommen.

Anmerkung: Das Modul (OSS SuperCartridge) schaltet beim Arbeiten intern Bänke um. Sobald das fertige Programm aber läuft, wird nur noch eine Bank benutzt (unter der der Speicher liegt), was die Umwandlung in ein RAM-basiertes 'Run Time Package' deutlich erleichterte.

3 Kompilieren eines Programms mit RunTime

Wie zuvor beschrieben, sucht Action! nach Symbolen immer zuerst in der aktuellen lokalen Library, dann in der globalen und schließlich in der System-Library im Modul. Diese Abfolge macht die Erzeugung eines 'RunTime-Action!-Programms' erst möglich.

Anmerkung: Erste Versionen von Action! (3.0 und 3.1) wiesen einen Fehler in der Divisionsroutine auf. Die (temporäre) Lösung bestand im Bereitstellen des Listings einer Action!-Routine, die aus einigen Blöcken in Maschinencode bestand. Bei Übernahme

86) Siehe Kapitel III, Abschnitt 2.11.
87) Das stimmt nur für DOS XL, OS/A+ sowie ATARI-DOS 2.x und dessen Derivate.
88) Siehe Kapitel VI, Die Action!-Library.

dieser Subroutine (direkt oder per INCLUDE) in ein Programm wurde der Compiler angewiesen, die neue Divisionsroutine anstelle der im Modul vorhandenen zu verwenden.

In vergleichbarer Weise besteht die RunTime aus PROCs und FUNCs (die ihrerseits hauptsächlich aus Maschinencode bestehen), die einem Programm hinzugefügt werden, sodass der Compiler ihre Namen in der globalen Symboltabelle anstelle der Namen der Modul-Routinen vorfindet.

3.1 Einfache Kompilierung

Die einfachste Art eine RunTime-Version eines Programms zu kompilieren besteht darin,

```
INCLUDE "D1:SYS.ACT"
```

als allererste Zeile in den Quelltext des Programms zu setzen.

Das File 'SYS.ACT' auf der RunTime-Diskette enthält Action!-Quellcode (zumeist als Codeblöcke) für ALLE Routinen in der System-Library im Modul. Wird dieses File am Anfang des Programms zuerst kompiliert, wird dem Compiler eine vollständige Liste der globalen Namen übergeben, die dann zuoberst in der Symboltabelle stehen und so Vorrang vor den gleichnamigen Routinen aus der System-Library haben.

Ein Anschauungsbeispiel ist das File 'SAMPLE.ACT' von der RunTime-Diskette. Wenn es in den Editor geladen ist, sieht man, wie das FILE 'SYS.ACT' per INCLUDE übernommen wird. Nachdem der Quelltext kompiliert wurde, lässt sich der Objektcode per W-Kommando auf Diskette schreiben (nicht auf die originale RunTime-Disk).

Das Ergebnis liegt bereits als 'SAMPLE.COM' auf der RunTime-Disk vor und kann von der DOS-Kommandozeile durch Eingabe des Namens 'SAMPLE' oder bei ATARI-DOS 2.x mit der L-Option geladen werden. In beiden Fällen sollte das Programm laufen und die erwarteten Resultate liefern.

3.2 Selektives Verwenden von Library-Routinen

Zusätzlich zur kompletten System-Library 'SYS.ACT' enthält die RunTime-Disk noch andere Library-Files:

SYSLIB.ACT	SYSIO.ACT	SYSGR.ACT
SYSMISC.ACT	SYSBLK.ACT	SYSSTR.ACT

Es gibt noch eine weiteres File namens 'SYSALL.ACT', das lediglich alle zuvor aufgeführten Files einfach per INCLUDE-Direktive einbindet. Das entspricht dem Einfügen von 'SYS.ACT' als Ganzes.

Jedes dieser Library-Files enthält einen Teil der ganzen RunTime-Library. Sie lassen sich ebenso per INCLUDE einbinden wie die gesamte 'SYS.ACT'. Dadurch kann auf nicht benötigte Routinen verzichtet werden.

Kommt ein Programm z.B. ohne Grafikeffekte aus, kann auf 'SYSGR.ACT' verzichtet werden. Die einzige Ausnahme ist 'SYSLIB.ACT'; diese Routinen werden praktisch immer benötigt. Eine vollständige Liste aller Routinen mit knappen Beschreibungen enthält das File 'SYS.DOC' auf der RunTime-Disk. Da nicht ohne Weiteres vorhersagbar ist, welche

Routinen tatsächlich von einem Programm benötigt werden, werden ausgelassene Runtime-Routinen durch namensgleiche Routinen aus der Library im Modul "ergänzt". Daher bietet es sich an ein Programm, das nicht mit der vollständigen 'SYS.ACT' kompiliert werden soll, sorgfältig nach Aufrufen von Library-Routinen zu überprüfen.

Für dieses Dilemma gibt es Abhilfe durch das Programm 'ST.ACT' auf der RunTime-Disk. Wird es kompiliert und gestartet, klinkt es sich in den Compiler ein und leistet nützliche Dienste: Beim Kompilieren eines Programms gibt es dessen Symboltabelle als Liste aus mit den Abhängigkeiten jeder einzelnen PROC und FUNC; zuerst alle lokalen Deklarationen, danach alle globalen Festlegungen. Voreingestellt ist die Ausgabe auf den Drucker. Weitere Informationen enthält das File 'ST.DOC' auf der RunTime-Disk.

Damit muss man zwar immer noch eine eventuell lange Liste daraufhin prüfen, welche Library-Routinen verwendet werden, was aber deutlich einfacher ist als die Prüfung des gesamten Quellcodes.

Schließlich noch ein Beispiel dafür, wie die selektive Verwendung von Library-Routinen den Platzbedarf im Speicher und auf Diskette verringert. Das Programm 'SAMPLE2.ACT' fügt per INCLUDE nur die Routinen ein, die es auch tatsächlich beim Ablauf benötigt. Wird es kompiliert und auf Diskette gespeichert, wird der Unterschied im Platzbedarf deutlich. Das bereits kompilierte Programm befindet sich ebenfalls als 'SAMPLE2.COM' auf der RunTime-Disk.

4 Kompilieren mit großen Symboltabellen

Wie bereits erläutert unterstützt Action! in der Voreinstellung bis zu 255 globale und 255 lokale Symbole[89]. Jedoch kann durch die Symboltabelle belegte Speicher niemals die Größe des per STSP ($0495) reservierten Speichers überschreiten.

Nachfolgend wird aufgezeigt, wie zwei der drei zuvor genannten Beschränkungen umgangen werden können. Derzeit existiert keine Möglichkeit in Action! mit mehr als 255 lokalen Symbolen zu arbeiten. Eigentlich ist das aber keine Beschränkung, denn sollten in einer PROC oder FUNC derartig viele Symbole verwendet werden, bietet es sich an, diese in mehrere Subroutinen aufzuteilen.

4.1 Vergrößern des Speichers für die Symboltabelle

Per Voreinstellung reserviert Action! 2KiB (2048 Bytes oder 8 Pages zu je 256 Bytes) an RAM für die Symboltabelle. Wird der Inhalt der Speicherstelle STSP ($0495) geändert, ändert sich damit die Größe des reservierten Speichers. Das muss allerdings vor dem Kompilieren erfolgen, da Action! Symboltabelle, Zeiger, etc. einrichtet, sobald aus dem Monitor das C-Kommando kommt.

Am einfachsten geht das aus dem Monitor heraus. Wird beispielsweise vor dem Kompilieren im Monitor die Anweisung

```
SET $495=12
```

eingegeben, werden dadurch 12 Pages, also 3KiB reserviert; danach dann kompilieren.

89) Kapitel VII, Abschnitt 2.1.3.

Alternativ kann man die SET-Direktive auch im Programm verwenden, wenn man weiß, wie groß die dafür benötigte Symboltabelle ist. Als allererste Zeile im Quellcode eingesetzt, bewirkt 'SET' ebenfalls eine entsprechende Reservierung an Speicher. Allerdings funktioniert das nicht beim ersten Kompilieren, da der bei Auslösung des C-Kommandos reservierte Speicher noch nicht ausreicht. Weil jedoch beim ersten Kompilieren die SET-Direktive ausgeführt wird(!), ist nun ausreichend Speicher für die Symboltabelle reserviert und erneutes Kompilieren dann erfolgreich.

4.2 Anzahl der globalen Symbole erhöhen

Die RunTime-Disk enthält ein File namens 'BIGST.ACT'. Wird es kompiliert und gestartet, stehen anschließend maximal 510 globale Symbole zur Verfügung.

Action! verfügt über ein Flag (BIGST; $04C4), an dem es erkennt, ob eine erweiterte globale Symboltabelle benötigt wird. Dafür nutzt Action! einen relativ einfach Mechanismus: Ist BIGST gesetzt, splittet Action! die globale Symboltabelle in zwei Teile unter Nutzung zweier separater Hash-Tabellen, ausschließlich basierend auf dem ersten Zeichen jedes Symbols. Dafür verwendet Action! die Speicherstelle FRSTCHAR ($04AD), deren Inhalt festlegt, bei welchem Zeichen die Tabelle gesplittet werden soll.

Zusätzlich zum Zeichen kann noch festgelegt werden, ob dabei nach Groß- und Kleinbuchstaben getrennt werden soll. Dazu wird das File 'BIGST.ACT' in den Editor geladen und der Wert für das Flag entsprechend der Anleitung darin gesetzt; zumeist wird aber die Voreinstellung ausreichen. Nachdem das File kompiliert und gestartet wurde, steht die große Symboltabelle so lange zur Verfügung, bis Action! erneut gebootet wird.

Übrigens verwendet Action! [STG2] ([$CE]) als Hash-Tabelle für die zusätzlichen globalen Symbole. STG2, BIGST und FRSTCHAR können auch direkt aus dem Monitor heraus gesetzt werden, aber mit 'BIGST.ACT geht es einfacher und sicherer.

5 Kompilieren an eine bestimmte Adresse

Normalerweise schreibt Action! den kompilierten Code direkt über den Editorpuffer in den Speicher, der seinerseits oberhalb des von Action! mit anderen Systemanteilen belegtem RAM liegt, die wiederum oberhalb von DOS abgelegt werden[90]. Nachfolgend werden Möglichkeiten erläutert, mit denen man Action! vorgeben kann, wohin der Code gespeichert werden soll.

5.1 Bestimmen der Speicheradresse für das Kompilat

Solange sich kein Programmtext im Editorpuffer befindet, ist der Bereich zwischen der Obergrenze der Systemanteile und der Untergrenze der Symboltabelle frei verfügbarer Speicher. Das Kompilat kann daher irgendwo in diesem Bereich abgelegt werden.

Vom Monitor aus lässt sich der freie Speicherbereich exakt bestimmen. Die Abfrage '? $E' liefert die Untergrenze. [APPMHI] enthält den Zeiger auf den Bereich für den kompilierten Code, also die Adresse, ab der das Kompilat im Speicher steht. Wenn noch kein Code kompiliert wurde, ist das die Adresse, ab der erst noch zu kompilierender Code in den Speicher geschrieben werden wird.

---

90) Siehe Anhang B.

Die Obergrenze dieses Bereichs wird im Monitor mit '? $B0' ermittelt. Speicherstelle $B0 (STBASE) enthält den 1-Byte-Wert, der die Nummer der Seite (Page) angibt, wo die Symboltabelle startet (minus 1 natürlich).

Schaut man sich nach dem Kompilieren den Wert in APPMHI (oder CODESIZE) an, sieht man, wie groß das kompilierte Programm ist. Belegt es nicht den gesamten freien Speicher, kann man es bei Bedarf im freien Speicher weiter nach oben verschieben.

Grundsätzlich müssen in Action! sowohl APPMHI als auch CODEBASE per SET auf die Startadresse des Codes gesetzt werden. Das lässt sich ganz einfach durch Einfügen von zwei SET-Anweisungen ganz am Anfang des Programms bewerkstelligen. Soll beispielsweise der Objektcode an Adresse $5000 abgelegt werden, lauten die ersten beiden Zeilen des Programmtextes:

```
SET $E=$5000
SET $491=$5000
```

Nicht vergessen: Der kompilierte Objektcode muss zwingend in den Speicher zwischen APPMHI und dem unteren Ende der Symboltabelle des Compilers passen.

5.2 Kompilieren mit einem Offset

Da das Modul, DOS sowie die Puffer und Tabellen von Action! festgelegte Speicherbereiche belegen, lässt sich der fertig kompilierte Code möglicherweise nicht an die gewollten Adressen ablegen. Soll ein Programm etwa das DOS ganz oder teilweise ersetzen, ist das nicht mehr so einfach mit SET machbar.

Aber keine Sorge, Action! wurde dafür ausgelegt, Code so zu kompilieren, dass er an anderen Adressen läuft als denen, an die das Kompilat abgelegt wurde.

Dafür wird ein einfacher Mechanismus angewendet: Die Speicherstelle CODEOFF ($B5) enthält einen 16-bit-Wert als Adressen-Offset. Der voreingestellte Wert ist null, weshalb der erzeugte Code dann auch an den Adressen läuft, an die er abgelegt wurde. Wird der Wert in CODEOFF geändert, sind wundersame Dinge möglich.

Immer wenn Action! eine Adresse für eine PROC, FUNC, Variable usw. berechnet, verwendet es dafür die in [APPMHI] festgelegte Speicherstelle. Wenn dann aber Action! einen Verweis auf so eine Adresse kompiliert, addiert es zu der Adresse den Wert in CODEOFF. Enthält z.B. ein Programm diesen Codeschnipsel:

```
SET $B5=$1000  ; CODEOFF auf 4KiB setzen
; angenommen, APPMHI enthält $4000
PROC P()
        ...
PROC Q()
        P()
        ...
```

Dann weiß der Compiler, dass die 'PROC P' an Adresse $4000 steht. Trifft er dann beim Kompilieren der 'PROC Q' auf den Verweis zu 'P', wird Code entsprechend dieser Beschreibung erzeugt:

```
JSR P+[CODEOFF]
   bzw.
JSR P+$1000
   bzw.
JSR $5000
```

Da Action! jeglichen Überlauf/Übertrag ignoriert, der durch Addieren von CODEOFF zu einer Adresse auftritt, ist Folgendes zulässig:

```
SET $B5=$F000
```

Dadurch werden dann im Endeffekt $1000 von jeder Adresse subtrahiert. Action! erlaubt keine negativen Compilerkonstanten; Ausnahme ist diese Vorgehensweise.

Angenommen, DOS sollte tatsächlich ersetzt werden, dann wäre ein Programm erforderlich, dass an Adresse $700 liefe. Hätte das verwendete DOS ein LOMEM von $2100, läge der ursprüngliche Wert von APPMHI bei etwa $2800 plus ein paar Bytes. Das entsprechende Action!-Programm würde dann mit den folgenden Zeilen beginnen:

```
SET $E=$2F00     ; nur um die Einstellung für
SET $491=$2F00   ; CODEOFF einfacher zu gestalten
SET $B5=$D800    ; entspricht $B5= -$2800
```

Und, siehe da, wenn der kompilierte Code untersucht würde, käme heraus, dass tatsächlich Code generiert worden wäre, der an Speicherstelle $700 liefe.

Wird nun das WRITE-Kommando des Monitors benutzt, werden beim Speichern automatisch Start- und Endadresse für das Objektcode-File so eingestellt, dass es beim späteren Laden von DOS aus an die Offset-Adresse geladen wird.

Sollte es warum auch immer nicht gewollt sein, dass der erzeugte Code an die fest-gelegte Adresse geladen wird, ist eine kleine Routine erforderlich, die den Code nach dem Laden an die festgelegte Adresse verschiebt und dann erst startet. Im obigen Beispiel würden beim Laden an Adresse $700 Teile des DOS direkt überschrieben werden, was mit Sicherheit sehr unangenehme Folgen hätte.

Um dies dennoch zu erreichen, gibt es eine einfache Vorgehensweise: Nach dem Kompilieren werden vom Monitor aus die Werte von CODEBASE und APPMHI ausgelesen und notiert. Nun Action! vom Monitor aus mit 'D' nach DOS verlassen, der erzeugte Code bleibt dabei im Speicher erhalten. Wird der Speicherbereich von CODEBASE bis APPMHI jetzt mit der SAVE-Funktion des DOS auf Disk abgespeichert, wird es anschließend an die Adresse geladen, an der es zuvor kompiliert wurde. Wieder wäre eine Routine erforderlich, die das Programm dann verschiebt und startet. Dafür kann z.B. eine Routine in Assembler an das Programm angehängt werden, die in Page 6 arbeitet.

Anmerkung: Diese Offset-Technik ist ebenfalls hilfreich, wenn ein Action!-Programm beinahe aber eben nicht zur Gänze in den 'freien' Speicher passt. Obwohl Arrays (nicht die kleinen BYTE-Arrays) quasi dynamisch hinter dem Ende des Programms abgelegt werden und dabei beispielsweise den Bereich für die Symboltabelle überschreiben könnten, beeinflussen sie nicht die Größe des kompilierten Programmcodes. Daher

können die von Action! vorbelegten und daher 'verlorenen' $700 Bytes an Speicher 'freigegeben' werden, wenn ein Offset von $F900 eingesetzt wird. Der gewonnene Speicherplatz ist nicht riesig (ca. 1700 Bytes), kann aber den entscheidenden Unterschied ausmachen.

5.3 Große Programmmodule in Maschinensprache

Da die Codegenerierung direkt vom Anwender gesteuert werden kann, ist es natürlich möglich, einen beliebigen Speicherbereich zu reservieren. Das setzt voraus, dass entsprechender Code für den bettreffenden Adressbereiche assembliert wird, eine Liste der Einsprungadressen und/oder Variablen erstellt wird für den Zugriff von Action! aus, und das Action!-Programm muss so erstellt werden, dass es den Speicherbereich des Assemblerprogramms meidet. Verweist das Action!-Programm über PROCs, FUNCs und Variablennamen auf Adressen in diesem Bereich, können die Routinen etc. in Assembler im Austausch mit Action!-Routinen verwendet werden. Nachfolgend dazu ein Beispiel:

```
Assembler

*=$3000

LSH3         ; FUNCTION: Linksverschiebung mit Argument 3
      ASL A
      ASL A
      ASL A ; 3-mal nach links schieben
      STA $A0 ; speichern dort wo Action! Das speichert
      LDA #0  ; ... Werte zurückgeben
      STA $A1
      RTS
MASK  .BYTE   1,2,4,8,16,32,64,128 ; Bitmaske setzen


Action!

BYTE FUNC LSH3=$3000 (BYTE N)
BYTE ARRAY MASK(0) = $300A
```

Die Routine und das Array ließen sich aufgrund ihrer geringen Größe auch direkt als Codeblöcke in das Action!-Programm einfügen. Jedoch für große, komplexe Prozesse ist das eine gut geeignete Methode.

6 Code für den ROM-Bereich kompilieren

Der vorherige Abschnitt lieferte auch gleich die Ideen, wie man den Action!-Compiler Code generieren lassen kann, der dann im Speicherbereich der Module von $A000 bis $BFFF läuft. Es wird also Code in einen sicheren Bereich im RAM kompiliert mit einem CODEOFF, der dafür sorgt, dass der Code dann im ROM-Bereich läuft.

Doch es gilt, noch ein kniffeliges Problem zu lösen. Was ist mit den Variablen? Beim Kompilieren verwendet Action! für alle Variablen, PROCs und FUNCs denselben Speicherbereich, wo sie gemischt abgelegt werden nach den Vorgaben von Action!. Es muss also dafür gesorgt werden, dass Action! Programm und Variablen getrennt ablegt.

6.1 RAM- und ROM-Variablen

Dafür gibt es eine einfache Lösung: allen Variablen Adressen zuweisen. Eine Deklaration wie

```
BYTE Temp = $D4
```

veranlasst Action! bei jeder Referenz zu 'Temp' auf Speicherstelle $D4 zu verweisen.

Es gibt eine zweite Gruppe an Variablen, die keiner Aufmerksamkeit bedürfen: diejenigen, die gar nicht 'variabel' sind. Wird eine Variable oder ein Array (oder String) initialisiert und der Inhalt im weiteren Verlauf nie geändert, sollten diese im ROM-Bereich liegen. Eine Deklaration wie

```
BYTE ARRAY Bits[0]=[1 2 4 8 16 32 64 128]
```

generiert und initialisiert ein Byte-Array mit 8 Elementen. Wird nie etwas in Bits(n) gespeichert, sollte das Array im ROM-Bereich liegen.

Die überwiegende Mehrzahl an Variablen fällt jedoch in keine der beiden zuvor genannten Kategorien. Das sind Variablen, deren Inhalte im Lauf des Programms geändert werden und für die der Compiler Speicher zuweisen soll.

Ursprünglich wurde Action! nicht dafür ausgelegt, Code zu erzeugen, der Programm und Variablen voneinander trennt. Doch die Funktionsweise der Compiler-Direktive SET eröffnet den Zugang zu einer fortschrittlichen Methode, die nachfolgend erläutert wird.

Dazu ist es vorteilhaft, das Listing des Files 'KALROM.ACT' von der RunTime-Diskette parat zu haben. Es ist eine etwas kleinere Version der Demo Kaleidoskop, die dafür ausgelegt wurde in den ROM-Bereich kompiliert zu werden.

Die beiden DEFINE-Direktiven am Anfang des Programms:

```
DEFINE RAM = "SET $682 = $E^
              SET $B5  = $C800
              SET $E   = $680^"

DEFINE ROM = "SET $680 = $E^
              SET $B5  = $5800
              SET $E   = $682^"
```

sorgen dafür zusammen mit den SET-Direktiven

```
SET $E=$6000   SET $491=$6000
SET $B5=$5800  SET $680=$5800
```

weiter unten im Listing. Das Programm wird dadurch an die Adresse $6000 kompiliert, auf die APPMHI und CODEBASE ursprünglich gesetzt wurden. Der Code wird aber so kompiliert, dass er an Adresse $A800 läuft, der Summe von APPMHI und CODEOFF.

Das vom Programm verwendete RAM wird an $5800 kompiliert (ursprünglicher Wert in der Speicherstelle $680, siehe unten) und wird dann, wenn der ROM-Code läuft, an

Adresse $2000 verschoben ($2000=$5800 + $C800; der alternative Wert für CODEOFF, wobei der durch die Addition erzeugte Überlauf ignoriert wird).

Anmerkung: Nach Fehlerliste Nr. 3 wird durch das zuvor beschriebenen DEFINE für ROM falscher Code erzeugt, wenn dabei lokale Variablen verwendet werden.

Die Funktionsweise dieser Methode erklärt sich wie folgt:

Die ersten SET-Direktiven (nicht die in den DEFINEs) sind vorbestimmte Werte, die Action! Code erzeugen lassen, der ab Adresse $A800 im Speicher liegt, wie zuvor erläutert. Erreicht der Compiler dann aber das Label 'RAM', führt er die darin festgelegten SET-Direktiven aus. Insbesondere wird der aktuelle Wert des Zeigers auf den Code (APPMHI) in einer 'freien' Speicherstelle ($682) per 'SET $682=$E' gesichert. Hier zeigt sich wieder, dass definierte Zeiger des Betriebssystems in einer SET-Direktive verwendet werden können. '$E' bedeutet ja nichts weiter als 'Inhalt der Speicherstelle $E'.

Die Ausführung der Definition für RAM bewirkt außerdem die Änderung von CODEOFF ($B5) und das Laden des Wertes aus einer weiteren 'freien' Speicherstelle ($682) nach APPMHI ($E; CODE). Die Speicherstelle $680 war ja zuvor nur zu diesem Zweck auf den Wert $5800 gesetzt worden.

Trifft der Compiler dann auf die Definition für 'ROM' und führt diese aus, passiert genau das Gegenteil: Der Wert von APPMHI wird in $680 gesichert, CODEOFF wird auf den erforderlichen Wert für das Erzeugen des Codes für ROM geändert und APPMHI wird auf den Wert aus $682 zurückgesetzt, der ja zuvor dort von der Definition für 'RAM' gesichert wurde.

Scheint sehr kompliziert zu sein, aber sobald die benötigten DEFINEs für 'ROM' und 'RAM' feststehen, ist der Rest ziemlich einfach.

Auf eines gilt es dabei aber zu achten, nämlich wann die DEFINEs für ROM und RAM im Programm verwendet werden. Grundsätzlich wird 'RAM' unmittelbar vor dem Festlegen von Variablen verwendet, die in RAM vorkommen sollen. Im Falle von lokalen Symbolen muss dann 'RAM' direkt vor deren Definition stehen und ROM direkt danach.

Dabei gibt es aber einen kleinen Haken. Nachdem auf ROM basierende Festlegungen für globale Variablen kompiliert wurden, muss 'RAM' im Programmtext eingesetzt werden, damit die Parameter und lokalen Variablen der nächsten PROC oder FUNC auch in RAM kompiliert werden können. Aufgrund der speziellen Weise, in der Action! sowohl Code als auch Verweise auf Adressen erzeugt, muss 'RAM' zwingend nach dem Schlüsselwort PROC oder FUNC stehen.

PROCs und FUNCs, an die keine Parameter übergeben werden und die keine lokalen Variablen aufweisen, können als vollständig im ROM liegend betrachtet werden. Aus Code-Blöcken bestehende PROCs und FUNCs, die nur Parameter in den CPU-Registern A, X und Y übergeben, dürfen in der Schreibweise '=*' beinhalten und erzeugen dann keinen Variablenspeicher.

6.2 Weitere Überlegungen

Sobald die Probleme für die Trennung von RAM- und ROM-Bereich angegangen werden, muss das Augenmerk noch auf einige andere Dinge gelegt werden, wenn Action!-Code für den ROM-Bereich erzeugt werden soll.

6.2.1 FOR-Schleifen

FOR-Schleifen können nicht in Action!-Code verwendet werden, der im ROM-Bereich laufen soll. Sobald Action! auf eine Anweisung der Form

```
FOR Schleifenvariable=Start TO Ende STEP Differenz
```

trifft, benötigt es Speicher zum Ablegen der Werte für Ende und Differenz. Sind diese Werte keine Konstanten, werden sie erst beim Programmablauf berechnet und dann irgendwo in den kompilierten Code geschrieben.

Das Problem lässt sich vermeiden, indem die FOR-Schleife in eine WHILE-Schleife umgewandelt wird:

```
Schleifenvariable=Start

WHILE Schleifenvariable <= Ende
DO
    ...
Schleifenvariable ==+ Differenz
OD
```

Ein wenig länger als die entsprechende FOR-Schleife, aber meist nicht weniger effizient.

6.2.2 PROC-Variablen

Mit Action! ist es möglich die Namen von PROCs in Ausdrücken zu verwenden einschließlich zuordnender Anweisungen zu einem PROC-Namen. Z.B. kann eine eigene Fehlerroutine etwa nach diesem Muster[91] geschrieben werden:

```
PROC HandleError()
       ...
RETURN

       ...

SaveError=Error
Error=HandleError

       ...
```

In Action! werden PROC-Namen dabei aufgrund eines unsichtbaren Prozesses wie folgt verarbeitet: Jede PROC oder FUNC wird so kompiliert, dass sie mit einem JMP-Befehl beginnt. Ziel dieses JMP ist standardmäßig das Byte unmittelbar nach dem JMP-Befehl, also der eigentliche Code der PROC oder FUNC.

91) Kapitel VI, Abschnitt 7.3.

Wird nun ein Wert einer PROC zugewiesen (wie zuvor in 'Error = HandleError'), modifiziert der erzeugte Code die letzten beiden Bytes des JMP-Befehls, nämlich die Zieladresse.

Liegt eine PROC oder FUNC im ROM, lässt sich das Ziel des JMP-Befehls natürlich nicht ändern. Ist das dennoch unbedingt erforderlich, bietet sich folgende Lösung dafür an:

```
PROC HandleError = $600 ()
        ; oder jede andere 'sichere' Adresse
BYTE Hjmp = $600 ; die gleiche Adresse
  ...
PROC RealHandler()
        ...
RETURN


  ...
MAIN()
      Hjmp = $4C ; ein Sprungbefehl
      HandleError = RealHandler
  ...
; und nun kann eine zuordnende Anweisung an 'HandleError' beliebig erfolgen
```

Das Wesentliche an diesem Trick ist die manuelle Festlegung des Sprungbefehls, wie sie in der ersten Zeile nach dem Aufruf von MAIN () vorgenommen wurde.

6.2.3 Das System-DEVICE von Action!

Viele der I/O-Routinen in der Library (sowohl Modul- als auch RunTime-Version) führen ihre Operationen über einen Kanal aus, der vom Inhalt der Speicherstelle namens DEVICE bestimmt wird. So nutzen z.B. PRINT() und INPUT() beide DEVICE.

Standardmäßig wird der Inhalt von DEVICE auf null initialisiert. Daher lässt sich der eingestellte Ausgabekanal einfach ändern, indem per OPEN ein anderer Kanal geöffnet und in DEVICE die neue Kanalnummer gesetzt wird.

Wird die RunTime verwendet, muss der Programmierer das Initialisieren von DEVICE übernehmen. Das kann z.B. mit

```
DEVICE = 0
```

erfolgen. Oder falls im geplanten Programm der Inhalt von DEVICE nie geändert werden soll, kann am Anfang eine globale Variable deklariert werden:

```
BYTE DEVICE = [0]
```

Weitere Hinweise dazu enthält das File SYS.DOC auf der RunTime_Disk.

6.2.4 Filenamen

Die Library im Modul ergänzt Filenamen automatisch um ein vorangestelltes 'D:', falls der Filenamen nicht mit einer Gerätebezeichnung wie 'D2:', 'P:', etc. beginnen sollte. Die Library der RunTime tut das nicht.

Daher unbedingt sicherstellen, dass bei Aufruf eines Filenamens die Gerätekennung korrekt mit angegeben wird.

VIII. Das Action!-Toolkit

1 Allgemeines

Das von OSS zum Programmiersystem Action! veröffentlichte Toolkit ist eine Art digitale Werkzeugkiste für Programmierer. Sie enthält in Action! geschriebene Routinen, mit deren Hilfe man die eigenen Programmierkenntnisse und -fertigkeiten verbessern kann. Aus diesem Grund wurde die Routinensammlung mit in das Handbuch übernommen. Eingearbeitet wurde die Toolkit-Version 3.

Zusätzlich zu den eigentlichen Toolkit-Routinen sind außerdem einige namensgleiche Dateien mit der Namenserweiterung 'DMn' enthalten ('n' steht dabei für eine Zahl). Dabei handelt es sich um Demos, die diese Routinen in einer kleinen Anwendung zeigen.

Des Weiteren sind einige Dateien mit der Namensendung '.DEM' Bestandteil des Toolkits. Dabei handelt sich um vollständige Programme zur Demonstration der Programmiermöglichkeiten in Action!. Die Programme sind nicht in diesem Handbuch abgedruckt.

Anmerkung: In den meisten Action!-Dateien kommen globale Variablen und Prozeduren vor, die das Unterstreichungszeichen '_' enthalten. Diese Variablen und Routinen sind interne Toolkit-Routinen und sollten weder aufgerufen noch bearbeitet werden, es sei denn, man weiß ganz genau, wozu sie dienen.

Die Routinen des Action!-Toolkit wurden ursprünglich auf einer Diskette herausgegeben, die nicht bootfähig ist, da kein DOS darauf enthalten ist.

Nachfolgend die Toolkit-Routinen in alphabetischer Reihenfolge. Die Vorstellung der einzelnen Routinen beinhaltet eine einführende Erläuterung sowie eine Erklärung der einzelnen Unterroutinen, begleitet von Anmerkungen, Hinweisen und Kommentaren.

Die Routinen können als Unterprogramme eigenen Programmen mithilfe der Direktive MODULE[92] hinzugefügt werden. Wenn sie nicht direkt in den Programmtext durch Hinzuladen im Editor eingefügt werden sollen, lassen sie sich beim Kompilieren per INCLUDE[93] einfügen.

Hinweis: Die Überschriften der Hauptabschnitte lauten auf die Dateinamen, unter denen die Toolkit-Routinen auf der Toolkit-Disk gespeichert sind.

92) Kapitel IV, Abschnitt 7.4.
93) Kapitel IV, Abschnitt 7.2.

2 ABS.ACT

Diese Routine liefert den absoluten Wert einer Integerzahl.

Zweck: Den absoluten Wert einer INT ermitteln.

Format: INT FUNC Abs(INT n)

Parameter: n →die Integerzahl, deren absoluter Wert ermittelt wird.

Beschreibung: Diese Funktion liefert den absoluten Wert der an sie übergebenen INT.

2.1 Programmtext

```
;************** ABS.ACT *************
; Funktion zum Auslesen des absoluten
; Wertes einer INT-Variablen.
;***********************************

INT FUNC Abs(INT n)
  IF n<0 THEN
    RETURN(-n)
  FI
RETURN(n)

MODULE
```

3 ALLOCATE.ACT

Mit den Routinen in dieser Datei können während des Programmablaufs Speicherbereiche zugewiesen und wieder freigesetzt werden. Dazu muss zuerst die Routine 'AllocInit' aufgerufen werden. 'AllocInit' benötigt eine globale CARD-Variable namens 'EndProg', welche die Adresse des jeweiligen Programmendes enthält. Diese erhält man direkt nach dem Kompilieren, wenn man im Monitor

```
SET EndProg=*<RETURN>
```

eingibt. Danach kann man das Programm starten.

Technische Anmerkung: Die Routinen für Zuweisung und Freigabe arbeiten mit einer 'Free List'. Diese Liste enthält Adresse und Größe jedes freien Speicherbereichs. 'Alloc' löscht einen neu belegten Speicherbereich aus dieser Liste und 'Free' setzt einen freigegebenen Bereich wieder auf diese Liste. Nachfolgend die Erläuterungen zu den einzelnen Routinen und am Ende des Abschnitts der gesamte Programmtext.

3.1 AllocInit

Zweck: Free List einrichten und Alloc-Routinen initialisieren.

Format: PROC AllocInit(CARD p)

Parameter: p die Adresse der ersten freien Speicherstelle.

Beschreibung: Diese Funktion wird zum Anlegen der 'Free List' benötigt, damit anschließend die Routinen 'Alloc' und 'Free' benutzt werden können.

Falls der Einsatz von Player/Missile-Grafik und/oder Bitmap-Grafik beabsichtigt ist, sollte zuvor die P/M-Grafik eingeschaltet werden und/oder das Programm sich in dem Grafikmodus befinden, der den meisten Speicher beansprucht. Danach kann erst 'AllocInit' aufgerufen werden, weil es den Speicher bis zu MEMTOP ($2E5) als freien Speicher betrachtet. Alternativ kann man auch den Wert in MEMTOP ändern.

Anmerkung: Aufgrund einer Änderung in den ALLOC-Routinen wird der für den Parameter eingegebene Wert ignoriert.

3.2 Alloc

Zweck: Reservieren eines Speicherbereichs einer festgelegten Größe und ausgeben der Startadresse desselben.

Format: CARD FUNC Alloc(CARD nBytes)

Parameter: nBytes Größe des reservierten Speichers in Bytes.

Beschreibung: Diese Routine ermöglicht die Reservierung eines Speicherbereichs von 'nBytes' Größe. Die Startadresse dieses Bereichs wird ausgegeben, sodass man z.B. auf diese Weise dem laufenden Programm Speicher für ein großes Array zuweisen kann, nachdem zuvor die Größe des Arrays z.B. so bestimmt wurde:

```
PROC Test()
  CARD size
```

```
  BYTE ARRAY bigarray

  Print(*Groesse des Arrays>> *)
  size=InputC()
  bigarray=Alloc(size)
RETURN
```

Anmerkung: Der kleinste freie Speicherbereich für eine Reservierung ist 3 Bytes.

3.3 Free

Zweck: Freigeben eines zuvor reservierten Speicherbereichs mithilfe der Alloc-Funktion.

Format: PROC Free(CARD target,nBytes)

Parameter: target Startadresse des freizugebenden Speichers.
nBytes Größe des freizugebenden Speichers.

Beschreibung: Mit dieser Routine wird ein von Alloc verwalteter Speicherbereich zurück auf die 'Free List' gesetzt.

3.4 PrintFreeList

Zweck: Ausgeben der 'Free List'.

Format: PROC PrintFreeList()

Parameter: keine

Beschreibung: Diese Routine gibt die 'Free List' aus, die hauptsächlich zum Diagnostizieren von Fehlerursachen verwendet werden sollte.

3.5 Programmtext

```
;*********** ALLOCATE.ACT ************
; Routinen zur dynamischen Speicher-
; zuweisung im laufenden Programm
;************************************

MODULE

DEFINE NULL="0"

TYPE BLOCK=[CARD size,next]

CARD MemLo=$2E7, MemHi=$2E5

BLOCK POINTER FreeList

;***********************************
; nBytes zuweisen und Startadresse
; dieses Speicherbereichs ausgeben
;***********************************
```

```
CARD FUNC Alloc(CARD nBytes)

  BLOCK POINTER last, current,
                target
  last=FreeList ; beginne am Anfang der Liste
  current=FreeList.next

  ; Liste nach Bereich ausreichender Groesse durchsuchen
  WHILE (current<>NULL) AND (current.size<nBytes)
    DO
      last=current
      current=current.next
    OD
  IF current=NULL THEN
    RETURN(NULL) ;couldn't find a block
  FI

  ; Fall 1 – Bereich hat exakt die gesuchte Groesse
  IF current.size=nBytes THEN
    last.next=current.next ; nur diesen Bereich
    target=current         ; aus der Free List entfernen
    RETURN(target)
  FI

  ;Fall 2 – Bereich ist groesser als noetig
  current.size==-nBytes ; von der Groesse abziehen
  target=current+current.size ;vom richtigen Ende her zuweisen
  target.size=nBytes
RETURN(target)

;*************************************
; Bereich freigeben und der Free List
; hinzufuegen
;*************************************
PROC Free(BLOCK POINTER target CARD nBYTES)

  BLOCK POINTER last, current

  target.size=nBytes
  last=FreeList
  current=FreeList.next

  ; Position zum Einfuegen des freigegebenen Bereichs suchen
  WHILE (current<>NULL) AND (current<target)
    DO
      last=current
      current=current.next
    OD
  ; Position gefunden, nun Bereich der Free List hinzufuegen
  IF last+last.size=target THEN
    last.size==+nBytes
```

```
    target=last
  ELSE
    target.next=current
    last.next=target
  FI

  ; prüfen auf richtige Einfuegung im Speicher
  IF target+target.size=current THEN
    target.size==+current.size
    target.next=current.next
  FI
RETURN

;*************************************
; Free List einrichten
;*************************************

PROC AllocInit(CARD useless)

  BLOCK POINTER p

  FreeList=EndProg
  p=EndProg+4
  FreeList.next=P
  p.next=NULL
  p.size=MemHi-P
RETURN

;*************************************
; Free List ausgeben
;*************************************

PROC PrintFreeList()

  BLOCK POINTER p

  P=FreeList.next
  WHILE p<>NULL
    DO
      PrintF("%H: %H %H%E",p,p.size,p.next)
      p=p.next
    OD
RETURN
```

4 CHARTEST.ACT

Diese Routinen führen verschiedene Teste und Veränderungen an Zeichen (Characters) durch. Die Routinen sind ziemlich unterschiedlich und umfassen

IsAlpha	Zeichenprüfung, ob es ein Buchstabe ist.
IsUpper	Zeichenprüfung, ob es ein Großbuchstabe ist.
IsLower	Zeichenprüfung, ob es ein Kleinbuchstabe ist.
IsDigit	Zeichenprüfung, ob es eine Ziffer ist.
ToUpper	Zeichenbearbeitung; auf Großbuchstabe ändern.
ToLower	Zeichenbearbeitung; auf Kleinbuchstabe ändern.

4.1 IsAlpha

Zweck: Ein einzelnes Zeichen prüfen, ob es ein Buchstabe ist.

Format: BYTE FUNC IsAlpha(BYTE c)

Parameter: c Das zu prüfende Zeichen.

Beschreibung: Diese Funktion prüft 'c' darauf, ob es alphabetisches Zeichen ist. Ist dies der Fall, wird eine 1 ausgegeben, andernfalls eine 0.

Programmtext:

```
;*************************************
; Pruefen ob 'c' ein Buchstabe ist
;*************************************

BYTE FUNC IsAlpha(BYTE c)

  IF ((c>='A) AND (c<='Z)) OR
       ((c>='a) AND (c<='z)) THEN
    RETURN(1)
  FI
RETURN(0)
```

4.2 IsUpper

Zweck: Ein einzelnes Zeichen prüfen, ob es ein Großbuchstabe ist.

Format: BYTE FUNC IsUpper(BYTE c)

Parameter: c Das zu prüfende Zeichen.

Beschreibung: Diese Funktion prüft 'c' darauf, ob es ein Großbuchstabe ist. Ist dies der Fall, wird eine 1 ausgegeben, andernfalls eine 0.

Programmtext:

```
;*************************************
; Pruefen ob 'c' Grossbuchstabe ist
;*************************************
BYTE FUNC IsUpper(BYTE c)
```

```
  IF (c>='A) AND (c<='Z) THEN
    RETURN(1)
  FI
RETURN(0)
```

4.3 IsLower

Zweck: Ein einzelnes Zeichen prüfen, ob es ein Kleinbuchstabe ist.

Format: BYTE FUNC IsLower(BYTE c)

Parameter: c Das zu prüfende Zeichen.

Beschreibung: Diese Funktion prüft 'c' darauf, ob es ein Kleinbuchstabe ist. Ist dies der Fall, wird eine 1 ausgegeben, andernfalls eine 0.

Programmtext:

```
;************************************
; Pruefen ob 'c' Kleinbuchstabe ist
;************************************

BYTE FUNC IsLower(BYTE c)

  IF (c>='a) AND (c<='z) THEN
    RETURN(1)
  FI
RETURN(0)
```

4.4 IsDigit

Zweck: Ein einzelnes Zeichen prüfen, ob es eine Ziffer ist.

Format: BYTE FUNC IsDigit(BYTE c)

Parameter: c Das zu prüfende Zeichen.

Beschreibung: Diese Funktion prüft 'c' darauf, ob es eine Ziffer ist. Ist dies der Fall, wird eine 1 ausgegeben, andernfalls eine 0.
Programmtext:

```
;*************************************
; Pruefen ob 'c' eine Ziffer ist (0-9)
;*************************************

BYTE FUNC IsDigit(BYTE c)

  IF (c>='0) AND (c<='9) THEN
    RETURN(1)
  FI
RETURN(0)
```

4.5 ToUpper

Zweck: Kleinbuchstabe in Großbuchstabe ändern.

Format: BYTE FUNC ToUpper(BYTE c)

Parameter: c Das zu ändernde Zeichen.

Beschreibung: Diese Funktion ändert 'c' von Kleinbuchstabe auf Großbuchstabe. Ist das Zeichen bereits ein Großbuchstabe oder kein Buchstabe, wird es unverändert ausgegeben. Die Funktion verwendet 'IsLower' zum Feststellen, ob es sich um einen Kleinbuchstaben handelt.

Programmtext:

```
;************************************
; 'c' in Grossbuchstabe aendern,
; falls moeglich
;************************************

BYTE FUNC ToUpper(BYTE c)

  IF IsLower(c) THEN
    c==-$20
  FI
RETURN(c)
```

4.6 ToLower

Zweck: Großbuchstabe in Kleinbuchstabe ändern.

Format: BYTE FUNC ToLower(BYTE c)

Parameter: c Das zu ändernde Zeichen.

Beschreibung:
Diese Funktion ändert 'c' von Großbuchstabe auf Kleinbuchstabe. Ist das Zeichen bereits ein Kleinbuchstabe oder kein Buchstabe, wird es unverändert ausgegeben. Die Funktion verwendet 'IsUpper' zum Feststellen, ob es sich um einen Großbuchstaben handelt.

Programmtext:

```
;************************************
; 'c' in Kleinbuchstabe aendern,
; falls moeglich
;************************************

BYTE FUNC ToLower(BYTE c)

  IF IsUpper(c) THEN
    c==+$20
  FI
RETURN(c)
```

5 CIRCLE.ACT

Diese Routine zum Zeichnen eines Kreises ist insofern besonders, als sie weder Sinus noch Cosinus berechnet und daher ziemlich schnell ist. Allerdings muss man bei ihrer Verwendung darauf achten, dass die Grenzen des Bildschirms in der jeweiligen Grafikstufe nicht überschritten werden. Man muss im Programm also entweder sicherstellen, dass der Kreis auf den Bildschirm passt, oder eine eigene Prüfroutine dafür einfügen.

Zweck: Zeichnen eines Kreises mit den Vorgaben Mittelpunkt, Radius und Farbe.

Format: PROC Circle(Int x BYTE y,r,c)

Parameter:
- x Die horizontale Koordinate des Kreismittelpunkts.
- y Die vertikale Koordinate des Kreismittelpunkts.
- r Der Radius des Kreises.
- c Die Farbe des Kreises.

Beschreibung: Mithilfe dieser Prozedur wird ein Kreis nach den vorgegebenen Werten für Mittelpunkt, Radius und Farbe gezeichnet. Die Systemvariable COLOR wird auf 'c' gesetzt, weshalb 'c' nicht die momentane Farbnummer sein sollte, wie sie in der Prozedur SetColor[94] verwendet wird. Stattdessen sollte es der Farbwert im gewählten Grafikmodus sein, der dasjenige SetColor-Register anspricht, das die Farbe enthält, die man verwenden will.

Programmtext:

```
;****************************************
; Schnelle Routine zum Zeichnen eines
; eines Kreises, die mit 8-facher
; Symmetrie arbeitet.
;****************************************

;****************************************
; Auslesen des absoluten Werts einer INT.
; Kann entfernt werden, wenn ABS.ACT
; zuvor INCLUDEt wurde.
;****************************************

INT FUNC Abs(INT n)

   IF n<0 THEN RETURN( -n ) FI
   RETURN( n )
```

(Fortsetzung auf nächster Seite)

```
;****************************************
; Die eigentliche Zeichenroutine.
;****************************************

PROC Circle(INT x,y,r,c)
```

94) Kapitel VI, Abschnitt 5.2.

```
  INT Phi,Phiy,Phixy,
      x1,y1

  Phi=0
  x1=r
  y1=0
  color=c
  DO
    Phiy=Phi+y1+y1+1
    Phixy=Phiy-x1-x1+1
    Plot(x+x1,y+y1) ; ┐
    Plot(x-x1,y+y1) ; |
    Plot(x+x1,y-y1) ; |
    Plot(x-x1,y-y1) ; ├ 8-fache Symmetrie
    Plot(x+y1,y+x1) ; ├ PLOTted die Punkte
    Plot(x-y1,y+x1) ; |
    Plot(x+y1,y-x1) ; |
    Plot(x-y1,y-x1) ; ┘
    Phi=Phiy
    y1=y1+1
    IF Abs(Phixy)+0<Abs(Phiy) THEN
      Phi=Phixy
      x1=x1-1
    FI
  UNTIL y1>x1
  OD
RETURN
```

6 CONSOLE.ACT

Die Routinen in dieser Datei ermöglichen die Ausführung einer eigenen Routine auf Drücken einer der CONSOLE-Tasten START, SELECT oder OPTION. Bevor man diese Möglichkeiten nutzen kann, muss man erst die Prozedur 'InitConsole' aufrufen. Nachdem das erfolgt ist, braucht man nur noch die Adresse einer Routine einer CONSOLE-Taste zuordnen. Das kann man wie folgt erledigen:

1. Eine Routine schreiben (braucht keine Parameter)
2. Die Prozedur 'Initconsole' aufrufen.
3. Bezeichnung der CONSOLE-Taste der Routine zuweisen.

6.1 Beispiel zur Verwendung

```
;*****************************
; Beispiel fuer die Verwendung
; von CONSOLE.ACT
;*****************************

INCLUDE "CONSOLE.ACT"

PROC DoStart()
  PrintE("START gedrueckt")
RETURN

PROC DoSelect()
  PrintE("SELECT gedrueckt")
RETURN

PROC DoOption()
  PrintE("OPTION gedrueckt")
RETURN

PROC Main()
  InitConsole()   ; Handler fuer CONSOLE-Tasten einrichten
  Start=DoStart   ; DoStart wenn START gedrueckt
  Select=DoSelect ; DoSelect wenn SELECT gedrueckt
  Option=DoOption ; DoOption wenn OPTION gedrueckt
  DO OD
RETURN
```

6.2 Programmtext

```
;*************************************
; Routinen zum Entprellen der CONSOLE-
; Tasten START,SELECT,OPTION mithilfe
; eines Timers und Aufruf einer fest-
; gelegten Routine durch Tastendruck.
;*************************************

MODULE

CARD Timer2=$21A,TempVec,
     Start,Select,Option
```

```
;**********************************
; Dummy zur Initialisierung
;**********************************

PROC RTS()
RETURN

;**********************************
; CONSOLE-Tasten-Routine abarbeiten
;**********************************

PROC DoConsole()

  [$6C TempVec]
RETURN

;**********************************
; Timer-Routine zum Entprellen und
; Vektor auf die CONSOLE-Tasten
;**********************************

PROC ConsoleTimer()

  BYTE console=$D01F

  DEFINE SaveTemps="[$A2 7 $B5 $A8
                   $48 $CA $10 $FA]",
         GetTemps="[$A2 0 $68 $95 $A8
                  $E8 $E0 8 $D0 $F8]"

  SaveTemps
  IF (console&1)=0 THEN
    TempVec=Start
    Timer2=30
    DoConsole()
    GetTemps
    RETURN
  FI
  IF (console&2)=0 THEN
    TempVec=Select
    Timer2=30
    DoConsole()
    GetTemps
    RETURN
  FI
  IF (console&4)=0 THEN
    TempVec=Option
    Timer2=30
    DoConsole()
    GetTemps
    RETURN
```

```
  FI
  Timer2=2
  GetTemps
RETURN

;***************************************
; Die CONSOLE-Routine auf Timer 2 setzen
;***************************************

PROC InitConsole()

  CARD Timer2Addr=$228

  Start=RTS
  Select=RTS
  Option=RTS
  Timer2Addr=ConsoleTimer
  Timer2=2
RETURN
```

7 IO.ACT

Mit diesen Routinen ist die Bearbeitung von Dateien aus einem Action!-Programm heraus bequem möglich. Folgende Optionen sind gegeben:

Datei umbenennen (rename file)
Datei löschen (erase file)
Datei sichern (protect file)
Datei entsichern (unprotect file)
Diskette formatieren[95] (format)
Datenblock von Medium einlesen (get data)
Datenblock auf Medium schreiben (put data)

Die ersten vier Optionen zur Bearbeitung von Dateien setzen voraus, dass mit der Datei die Gerätekennung in der Form 'Dn:' (n=1-8[96]) angegeben wird. Andernfalls tritt der Fehler 130 (nonexistent device - nicht vorhandenes Gerät) auf. Die ersten fünf Routinen verwenden die von BASIC her bekannten XIO-Kommandos[97].

7.1 Rename

Zweck: Umbenennen einer Datei.

Format: PROC Rename(BYTE ARRAY filename)

Parameter:	Dn:	Gerätekennung
	filename.ext	alter Dateiname
	<SPACE> oder <,>	Leerzeichen oder Komma
	filename.ext	neuer Dateiname

Beschreibung: Die Routine ändert den Namen einer Datei, was z.B. wie folgt angewendet werden muss

```
Rename("D1:TEMP1.ACT TEMP.ACT")
```

Damit wird der Name der Datei 'TEMP1.ACT' auf Laufwerk 'D1:' in 'TEMP.ACT' geändert.

Programmtext:

```
;************************************
; Datei auf Laufwerk umbenennen.
;************************************

PROC Rename(BYTE ARRAY filename)
  XIO(5,0,32,0,0,filename)
RETURN
```

95) Führt bei SpartaDOS X in das Menü 'SpartaDOS Formatter'
96) 9 für MyDOS, SpartaDOS 3.x und SpartaDOS X 4.2x; 15 für SpartaDOS X 4.4x
97) Kapitel VI, Abschnitt 4.3.

7.2 Erase

Zweck: Löschen einer Datei.

Format: PROC Erase(BYTE ARRAY filename)

Parameter: Dn: Gerätekennung
filename.ext Dateiname der zu löschenden Datei.

Beschreibung: Diese Prozedur löscht eine Datei von einem Laufwerk und muss wie folgt angewendet werden:

```
Erase("D2:JUNK.ACT")
```

Dadurch wird die Datei 'JUNK.ACT' auf Laufwerk 'D2:' gelöscht.

Programmtext:

```
;*************************************
; Datei vom Laufwerk löschen
;*************************************

PROC Erase(BYTE ARRAY filename)
  XIO(5,0,33,0,0,filename)
RETURN
```

7.3 Protect

Zweck: Sichern einer Datei.

Format: PROC Protect(BYTE ARRAY filename)

Parameter: Dn: Gerätekennung
filename.ext Dateiname der zu sichernden Datei.

Beschreibung: Diese Prozedur sichert eine Datei auf einem Laufwerk und muss wie folgt angewendet werden:

```
Protect("D2:JUNK.ACT")
```

Dadurch wird die Datei 'JUNK.ACT' auf Laufwerk 'D2:' gesichert, indem ein Schreibschutz aktiviert wird. Mithilfe von

```
Protect("D:*.*")
```

werden alle Dateien auf Laufwerk 'D:'[98] mit einem Schreibschutz versehen.

Programmtext:

(folgt auf der nächsten Seite)

98) Konvention des verwendeten DOS beachten.

```
;************************************
; Datei auf Laufwerk sichern
;************************************

PROC Protect(BYTE ARRAY filename)

  XIO(5,0,35,0,0,filename)
RETURN
```

7.4 UnProtect

Zweck: Entsichern einer Datei.

Format: PROC UnProtect(BYTE ARRAY filename)

Parameter: Dn: Gerätekennung
filename.ext Dateiname der zu entsichernden Datei.

Beschreibung: Diese Prozedur entsichert eine Datei auf einem Laufwerk und muss wie folgt angewendet werden:

```
UnProtect("D3:REVISION.ACT")
```

Dadurch wird die Datei 'REVISION.ACT' auf Laufwerk 'D3:' entsichert, da der Schreibschutz aufgehoben wird. Durch

```
UnProtect("D9:*.*")
```

wird bei allen Dateien auf Laufwerk 'D9:' der Schreibschutz entfernt.

Programmtext:

```
;************************************
; Unprotect a disk file
;************************************

PROC UnProtect(BYTE ARRAY filename)

  XIO(5,0,36,0,0,filename)
RETURN
```

7.5 Format

Zweck: Formatieren eines Mediums.

Format: PROC Format(BYTE ARRAY DriveSpec)

Parameter: Dn: Gerätekennung des Laufwerks, welches das zu formatierende Medium enthält.

Beschreibung: Mit dieser Prozedur kann ein Medium wie folgt formatiert werden:

```
Format("D5:")
```

Dadurch wird das Medium in Laufwerk 'D5:' in SD[99] formatiert, es sei denn, es verfügt über einen aktiven elektronischen[100] oder mechanischen Schreibschutz.

Programmtext:

```
;************************************
; Medium formatieren
;************************************

PROC Format(BYTE ARRAY DriveSpec)

  XIO(5,0,253,0,0,DriveSpec)
RETURN
```

7.6 BGet

Zweck:	Einlesen eines Blocks von Binär- oder Textdaten vom angegebenen Gerät.	
Format:	CARD FUNC BGet(BYTE chan CARD addr,len)	
Parameter:	chan	ist eine zulässige Kanalnummer (0-7).
	addr	Adresse im Speicher, ab der die Daten abgelegt werden sollen.
	len	Die Anzahl einzulesender Daten.

Beschreibung: Mit dieser Funktion kann man einen Block an Daten einlesen, wobei sie die Anzahl der tatsächlich eingelesenen Daten ausgibt. Diese kann ggf. geringer sein, wenn das Ende der Datei (EOF) erreicht wurde, bevor die mit 'len' vorgegebene Anzahl an Daten eingelesen wurde.

Programmtext:

```
;************************************
; Datenblock einlesen
;************************************

CARD FUNC BGet(BYTE chan CARD addr,len)

  CARD temp

  temp=Burst(chan,7,addr,len)
RETURN (temp)
```

Anmerkung: Diese Funktion verwendet ebenso wie die nachfolgende Funktion 'Bput' eine schnelle Block-I/O-Routine namens 'burst', die nach diesen beiden Funktionen am Ende des Abschnitts zu finden ist.

---

99) SD – Single Density, 90KByte
100) SpartaDOS 2.x/3.x, Kommando 'Protect Disk'

7.7 BPut

Zweck: Ausgeben eines Blocks von Binär- oder Textdaten auf das angegebene Gerät.

Format: PROC BPut(BYTE chan CARD addr,len)

Parameter:
chan ist eine zulässige Kanalnummer (0-7).
addr Adresse im Speicher, ab der die auszugebenden Daten zu finden sind.
len ist die Anzahl auszugebender Daten.

Beschreibung: Mit dieser Funktion kann man einen Block an Daten ausgeben.

Programmtext:

```
;************************************
; Datenblock ausgeben
;************************************
PROC BPut(BYTE chan CARD addr,len)

  Burst(chan,11,addr,len)
RETURN
```

Anmerkung: Diese Funktion verwendet ebenso wie die vorherige Funktion 'BGet' eine schnelle Block-I/O-Routine namens 'burst', die nun folgt.

7.8 Burst I/O

Mit dieser Funktion werden die notwendigen Parameter direkt in die IOCB-Register geschrieben; anschließend wird die CIO aufgerufen.[101]

Programmtext:

```
;************************************
; Schnelle (Block) I/O Routine
; per CIO-Aufruf
;************************************

PROC CIO=$E456(BYTE areg,xreg)

;************************************

CARD FUNC Burst(BYTE chan,mode,
                CARD addr,buflen)

  TYPE IOCB=[BYTE id,num,cmd,stat
             CARD badr,padr,blen
             BYTE a1,a2,a3,a4,a5,a6]

  IOCB POINTER iptr
```

101) Kapitel IV, Abschnitt 9.4.

```
  chan==&$07
  iptr=$340+(chan LSH 4)
  iptr.cmd=mode
  iptr.blen=buflen
  iptr.badr=addr
  CIO(0,chan LSH 4)
RETURN(iptr.blen)
```

8 JOYSTIX.ACT

Diese beiden folgenden Routinen ermöglichen die Abfrage der Bewegungsrichtungen bei Joysticks und sind einfacher zu benutzen als die entsprechende Library-Routine.

8.1 HStick

Zweck: Abfragen der Horizontalwerte des angegebenen Joysticks.

Format: INT FUNC HStick(BYTE port)

Parameter: port die Port-Nummer des Joysticks, dessen Horizontalwerte abge fragt werden sollen.

Beschreibung: Diese Routine liest die Werte der Joystickregister wie folgt aus:

-1 horizontale Bewegung nach links
0 keine horizontale Bewegung
1 horizontale Bewegung nach rechts

Programmtext:

```
;*************************************
; horizontale Bewegung des Joysticks
;            -1 # 0 # 1
;*************************************

INT FUNC HStick(BYTE port)

  BYTE ARRAY ports(4)=$278

  INT ARRAY value(4)=[0 1 $FFFF 0]

  port==&3
RETURN (value((ports(port)&$C) RSH 2))
```

8.2 VStick

Zweck: Abfragen der Vertikalwerte des angegebenen Joysticks.

Format: INT FUNC VStick(BYTE port)

Parameter: port die Port-Nummer des Joysticks, dessen Vertikal werte abgefragt werden sollen.

Beschreibung: Diese Routine liest die Werte der Joystickregister wie folgt aus:

-1 vertikale Bewegung nach oben
0 keine vertikale Bewegung
1 vertikale Bewegung nach unten

Programmtext:

```
;*************************************
; vertikale Bewegung des Joysticks
;           -1 # 0 # 1
;*************************************

INT FUNC VStick(BYTE port)

  BYTE ARRAY ports(4)=$278

  INT ARRAY value(4)=[0 1 $FFFF 0]

  port==&3
RETURN (value(ports(port)&3))
```

9 PMG.ACT

Mithilfe dieser Routinen lassen sich die Grafikfähigkeiten des ATARIs im Bereich Player/Missile (P/M) leicht in eigene Programme einbauen. Eine zusammenfassende Übersicht der P/M-Routinen:

PMGraphics	P/M ein-/ausschalten
PMColor	Farbe der P/M setzen
PMAdr	Adresse eines P/M ausgeben
PMClear	Speicherbereich eines P/M leeren
PMMove	P/M bewegen
PMCreate	P/M erstellen
PMHit	die Kollisionsregister der P/M abfragen
PMHitClr	Kollisionsregister zurücksetzen
PMHPos	horizontale Position eines P/M feststellen
PMVPos	vertikale Position eines P/M feststellen
Graphics	veränderte Grafikprozedur[102]

Einigen Routinen verwenden den Parameter 'n', der sich auf die jeweilige P/M-Nummer bezieht. Für 'n' gelten folgende Zuweisungen:

0 - Player 0	4 - Missile 0
1 - Player 1	5 - Missile 1
2 - Player 2	6 - Missile 2
3 - Player 3	7 - Missile 3

In manchen Fällen sind nur die Werte 0-3 zulässig bzw. sinnvoll.

Die von den Routinen gemeinsam genutzten Variablen lauten wie folgt:

```
BYTE PM_Mode=[0],
     PMHitClr=$D01E

CARD PM_BaseAdr

BYTE ARRAY
      PM_Hpos(8)=$D000,
      PMHpos(8)=[0 0 0 0 0 0 0 0],
      PMVpos(8)=[0 0 0 0 0 0 0 0],
      PM_MisMask(4)=[$FC $F3 $CF $3F],
      PM_Width(5)=$D008

CARD ARRAY
      PM_BSize=[0 $100 $80],
      PM_Waste=[0 768 384]
```

Nun zu den einzelnen Routinen.

102) Kapitel VI, Abschnitt 5.1.

9.1 PMGraphics

Zweck: Die P/M-Grafik ein- oder ausschalten.

Format: PROC PMGraphics(BYTE mode)

Parameter: mode legt den P/M-Modus fest.

Beschreibung: Diese Routine entspricht weitgehend der Grafikroutine in der Action!-Library, nur dass diese die P/M-Grafik steuert. Die Modi sind wie folgt:

- 0 P/M ausschalten
- 1 P/Ms in einzeiliger Auflösung
- 2 P/Ms in zweizeiliger Auflösung

Programmtext:

```
;*************************************
; Routine zum Ein-/ausschalten der
; P/M. Wenn Speicher zugewiesen wird,
; arbeitet sie von HiMem abwaerts.
;*************************************

PROC PMGraphics(BYTE mode)

  BYTE DMACtl=$22F,   ; P/M Register setzen
       Priority=$26F,
       GRACtl=$D01D,
       PMBase=$D407,
       GraphP0=$D00D

  CARD HiMem=$2E5,
       OldHimem,
       AppMHi=$E

  CARD ARRAY PM_AdrMask=[0 $F800 $FC00],
             PM_MemSize=[0 $800 $400]

  ; sicherstellen eines zulaessigen Modus
  ; - andernfalls Grafikaufruf ignorieren.
  IF mode > 2 THEN
    mode = 0
  FI

  Zero(PM_Hpos,8)  ; Bildschirm saeubern
  Zero(PMHpos,8)
  Zero(PMVpos,8)
  Zero(PM_Width,5)
  Zero(GraphP0,5)
  IF PM_Mode#0 THEN
    Himem=OldHiMem
    DMACtl=$22
```

```
    GRACtl=0
  FI
  IF mode=0 THEN   ; P/M ausschalten
    DMACtl=$22
    GRACtl=0
  ELSE
    IF mode=1 THEN ; einzeilige Aufloesung
      DMACtl=$3E
    ELSE           ; zweizeilige Aufloesung
      DMACtl=$2E
    FI
    OldHiMem=HiMem
    PM_BaseAdr=(HiMem-PM_MemSize(mode)-$80)&PM_AdrMask(mode)
    PMBase=PM_BaseAdr RSH 8
    IF PM_BaseAdr<AppMHi THEN
      DMACtl=$22
      GRACtl=0
      RETURN
    FI
    HiMem=PM_BaseAdr+PM_Waste(mode)
    Priority==&$C0%1
    GRACtl=3
  FI
  PM_Mode=mode   ; Modus speichern
RETURN
```

9.2 PMColor

Zweck: Die P/M-Grafik ein- oder ausschalten.

Format: PROC PMColor(BYTE n,hue,lum)

Parameter:
- n — Nummer des Players (0-3).
- hue — die Farbe für den Player
- lum — die Helligkeit für den Player

Beschreibung: Diese Routine entspricht der SetColor-Routine in der Action!-Library[103]. Sie erzeugt exakt die gleichen Farben, nur dass hiermit die Farbe für einen Player und nicht für den Hintergrund gesetzt wird.
Programmtext:

```
;************************************
; Farbe eines P/M setzen.
;************************************

PROC PMColor(BYTE n,hue,lum)

  BYTE ARRAY PM_Color(4)=$2C0

  n==&3
```

103) Kapitel VI, Abschnitt 5.2.

```
  PM_Color(n)=(hue LSH 4) % (lum & $0F)
RETURN
```

9.3 PMAdr

Zweck: Adresse des Speicherbereichs zum jeweiligen P/M auslesen.

Format: CARD FUNC PMAdr(BYTE n)

Parameter: n Nummer des Players (0-3).

Beschreibung: Die Routine liefert die Startadresse des Speicherbereichs zu dem Player, der mit 'n' angegeben wurde. Da die Missiles alle den gleichen Speicherbereich belegen, liefern die 'n'-Werte 4-7 alle die gleiche Adresse.

Programmtext:

```
;***********************************
; Startadresse zum Speicherbereich
; eines P/M auslesen.
;***********************************

CARD FUNC PMAdr(BYTE n)

  n==&7
  IF n>=4 THEN
    n=0
  ELSE
    n==+1
  FI
RETURN(PM_BaseAdr+PM_Waste(PM_Mode)+(n*PM_BSize(PM_Mode)))
```

9.4 PMClear

Zweck: Den Speicherbereich des jeweiligen P/M leeren.

Format: PROC PMClear(BYTE n)

Parameter: n P/M-Nummer

Beschreibung: Die Prozedur setzt alle Bytes im Speicherbereich des durch 'n' festgelegten P/M auf null. Handelt es sich dabei um eine Missile, wird nur der Teil des Speichers auf null gesetzt, welcher der Missile zugewiesen ist.

Programmtext:

```
;***********************************
; Speicherbereich eines P/M
; auf null setzen.
;***********************************

PROC PMClear(BYTE n)
```

```
  CARD ctr

  BYTE ARRAY PlayAdr

  n==&7
  playAdr=PMAdr(n)
  IF n<4 THEN
    Zero(PlayAdr,PM_BSize(PM_Mode))
  ELSE
    n==-4
    FOR ctr=0 TO PM_BSize(PM_Mode)-1
      DO
      PlayAdr(ctr)==&PM_MisMask(n)
      OD
  FI
RETURN
```

9.5 PMMove

Zweck: Den Speicherbereich des jeweiligen P/M leeren.

Format: PROC PMMove(BYTE n,x,y)

Parameter:
- n P/M-Nummer
- x Horizontalposition, an die der P/M bewegt werden soll.
- y Vertikalposition, an die der P/M bewegt werden soll.

Beschreibung: Die Prozedur ermöglicht die einfache und schnelle Bewegung eines P/M. Man braucht nur die Nummer des P/M sowie die X/Y-Position anzugeben, an die er verschoben werden soll.

Programmtext:

```
;***********************************************
; P/M zu einer absoluten (x,y) Position bewegen
;***********************************************

PROC PMMove(BYTE n,x,y);

  CARD i

  BYTE yOffset,
       plLength,
       mask1,mask2

  INT deltaY

  BYTE ARRAY temp(256),PlPtr

  IF PM_Mode=0 THEN
    RETURN
  FI
```

```
  n==&7
  deltaY = y
  deltaY = deltaY-PMVpos(n)
  IF deltaY=0 THEN
    PM_Hpos(n)=x
    PMHpos(n)=x
    RETURN
  FI
  plPtr=PMAdr(n)
  plLength=PM_BSize(PM_Mode)
  IF deltaY>=0 THEN
    yOffset=deltaY
  ELSE
    yOffset=plLength+deltaY
  FI
  IF n<4 THEN
    mask1=255
    mask2=0
  ELSE
    mask2=PM_MisMask(n&3)
    mask1=mask2!$FF
  FI

;-------------------------------------
; Dieser Teil des Codes
;
; FOR i=0 to plLength-1
;  DO
;   temp(i)=plPtr(i)&mask1
;  OD
;
; wurde durch den nachfolgenden
; Codeblock ersetzt, um die
; Geschwindigkeit zu erhoehen.
;-------------------------------------

[ $A0 0         ; LDY #0
  $AD PlPtr     ; LDA PLPTR
  $85 $A0       ; STA $A0
  $AD PlPtr+1   ; LDA PLPTR+1
  $85 $A1       ; STA $A1
;LOOP
  $B1 $A0       ; LDA ($A0),Y
  $2D mask1     ; AND MASK1
  $99 temp      ; STA TEMP,Y
  $C8           ; INY
  $CC plLength ; CPY PLLENGTH
  $D0 $F2       ; BNE LOOP
]
```

```
;-------------------------------------
; Dieser Teil des Codes
;
; FOR i=0 to plLength-1
;  DO
;   plPtr(yOffset)==&mask2 %temp(i)
;   yOffset==+1
;   IF yOffset>=plLength THEN
;     yOffset=0
;   FI
;  OD
;
; wurde durch den nachfolgenden
; Codeblock ersetzt, um die
; Geschwindigkeit zu erhoehen.
;-------------------------------------

[ $A2 0        ; LDX #0
  $AC yOffset  ; LDY YOFFSET
  $AD PlPtr    ; LDA PLPTR
  $85 $A0      ; STA $A0
  $AD PlPtr+1  ; LDA PLPTR+1
  $85 $A1      ; STA $A1
;LOOP
  $B1 $A0      ; LDA ($A0),Y
  $2D mask2    ; AND MASK2
  $1D temp     ; ORA TEMP,X
  $91 $A0      ; STA ($A0),Y
  $C8          ; INY
  $CC plLength ; CPY PLLENGTH
  $D0 2        ; BNE ISLOW
  $A0 0        ; LDY #0
;ISLOW
  $E8          ; INX
  $EC plLength ; CPX PLLENGTH
  $D0 $E8      ; BNE LOOP
]

  PMVpos(n)=y
  PM_Hpos(n)=x
  PMHpos(n)=x
RETURN
```

9.6 PMCreate

Zweck: Einfache Erzeugung eines P/M.

Format: PROC PMCreate(BYTE n BYTE ARRAY pm BYTE len,width,x,y)

Parameter:

n	P/M-Nummer
pm	das Array mit den Daten für die Figur des P/M.
len	die Länge des Arrays 'pm'.
width	die Breite des Players.
x	horizontale Angabe zur Startposition des P/M.
y	vertikale Angabe zur Startposition des P/M.

Beschreibung: Die Routine ermöglicht die Erzeugung eines P/M. Man übergibt folgende P/M-Daten: Nummer, Name des Arrays, das die Figur enthält, die Länge des Arrays, Breite (1=einfache, 2=doppelte, 4=vierfache) und X/Y-Angabe zur Startposition.

Programmtext:

```
;*************************************
; P/M erzeugen
;*************************************

PROC PMCreate(BYTE n BYTE ARRAY pm BYTE len,width,x,y)

  BYTE i,mask,temp,ntemp,
       oldwidth=[0]

  BYTE ARRAY plPtr,
             miswidth=[0 1 0 3]

  n==&7
  IF n<4 THEN
    mask=0
  ELSE
    ntemp=n&3
    mask=PM_MisMask(ntemp)
  FI
  plPtr=PMAdr(n)
  FOR i=0 to len-1
   DO
    plPtr(i+y)==&mask%pm(i)
   OD
  width==-1
  IF n<4 THEN
    PM_Width(n)=width
  ELSE
    temp=(miswidth(width) LSH (ntemp LSH 1))
    oldwidth==&mask%temp
    PM_Width(4)=oldwidth
  FI
  PM_Hpos(n)=x
```

```
  PMHpos(n)=x
  PMVpos(n)=y
RETURN
```

9.7 PMHit

Zweck: Feststellen, ob ein vorgegebener P/M mit einem bestimmten Player oder Playfield[104] kollidiert ist.

Format: BYTE FUNC PMHit(BYTE n,cnum)

Parameter:
- n — P/M-Nummer
- cnum — P/M oder Playfield, das auf Kollision geprüft wird.

Beschreibung: Mit dieser Routine kann man feststellen, ob ein vorgegebener P/M mit einem bestimmten anderen P/M oder Playfield kollidiert ist. Ist eine Kollision erfolgt, wird eine '1' ausgegeben, andernfalls eine '0'. Die Werte für 'n' wurde bereits am Beginn des Abschnitts 8 erläutert. Die Werte für 'cnum' haben folgende Bedeutung:

0 - Player 0	8 - Playfield 0
1 - Player 1	9 - Playfield 1
2 - Player 2	10 - Playfield 2
3 - Player 3	11 - Playfield 3

Die Playfield-Nummern 0-3 sind die gleichen, die von der Library-Routine SetColor[105] zum Setzen der Playfield-Farben verwendet werden.

Programmtext:

```
;*************************************
; Pruefung auf P/M-Kollision
;*************************************

BYTE FUNC PMHit(BYTE n,cnum)

  BYTE ARRAY pmtopf(8)=$D000,
             pmtop(8)=$D008

  n==&7
  IF n<4 THEN
    n==+4
  ELSE
    n==-4
  FI
  IF cnum<4 THEN
    RETURN ((pmtop(n) RSH cnum) & 1)
  ELSE
    cnum==&3
    RETURN ((pmtopf(n) RSH cnum) & 1)
```

104) ABBUC-Edition des ATARI-Profibuchs.
105) Kapitel VI, Abschnitt 5.2.

```
  FI
RETURN (0)
```

9.8 PMHitClr

Zweck: Löscht die Inhalte der P/M-Kollisionsregister.

Format: PMHitClr=0

Parameter: keine

Beschreibung: Mit der zuordnenden Anweisung PMHitClr=0 kann man die Inhalte der P/M-Kollisionsregister löschen. Das sollte erfolgen, bevor man etwas ausführen lässt, das zu einer Kollision führen kann, z.B. ein 'PMMove', oder wenn noch Werte von vorherigen Kollisionen in den Registern vorhanden sind.

9.9 PMHPos

Zweck: Den aktuellen horizontalen Positionswert eines P/M feststellen.

Format: BYTE ARRAY PMHPos(8)

Parameter: Die Nummer des Elements aus dem Array (entspricht der P/M-Nummer).

Beschreibung: Durch Zugriff auf ein Element aus diesem Array[106] kann man den aktuellen horizontalen Positionswert eines beliebigen P/M feststellen. Man verwendet einfach die P/M-Nummer als Element des Arrays. 'PMHPos(3)' z.B. liefert die horizontale Position von Player 3. Die Werte in dem Array sollten nicht verändert werden.

9.10 PMVPos

Zweck: Den aktuellen vertikalen Positionswert eines P/M feststellen.

Format: BYTE ARRAY PMVPos(8)

Parameter: Die Nummer des Elements aus dem Array (entspricht der P/M-Nummer).

Beschreibung: Durch Zugriff auf ein Element aus diesem Array[71] kann man den aktuellen vertikalen Positionswert eines beliebigen P/M feststellen. Man verwendet einfach die P/M-Nummer als Element des Arrays. 'PMVPos(5)' z.B. liefert die vertikale Position von Missile 1. Die Werte in dem Array sollten nicht verändert werden.

9.11 Graphics

Zweck: Abschalten der P/M-Grafik, sobald der Grafikmodus der Bitmap-Grafik geändert wird.

Format: PROC Graphics(BYTE mode)

Parameter: mode Nummer des Grafikmodus[107].

106) Kapitel VII, Abschnitt 8.
107) ABBUC-Edition des ATARI Profibuchs

Beschreibung: Diese Prozedur macht nichts weiter als die P/M-Grafik abzuschalten, sobald der Modus der Bitmap-Grafik geändert wird, und die Grafik-Routine der Action!-Library[108] zu ersetzen. Diese Routine ist notwendig, da der für die P/M-Grafik zugewiesene Speicher direkt unterhalb des Bildschirmspeichers liegt. Durch eine Änderung des Grafikmodus könnte der P/M-Speicherbereich teilweise überschrieben werden. Wechselt man im Programm nur zwischen Grafikmodi, die einen gleich großen Speicherbedarf aufweisen, kann man diese Prozedur im Programmtext auskommentieren und so die Grafik unverändert lassen.

Programmtext:

```
;****************************************
; Ersetzt die Library-Routine Graphics
; so, dass die P/Ms zurückgesetzt werden,
; wenn der Bitmap-Modus geaendert wird.
;****************************************

PROC Graphics(BYTE mode)
  PMGraphics(0)
  Close(6)
  Open(6,"S:",(mode&$F0)!$1C,mode)
RETURN
```

108) Kapitel VI, Abschnitt 5.1.

10 PRINTF.ACT

Bei den nächsten beiden Prozeduren handelt es sich um Erweiterungen der Library-Routine PrintF[109]. Damit lassen sich sowohl Feldgröße und Ausrichtung als auch die Art der Datenausgabe steuern.

Diese Routinen sind interne Bestandteile der PrintF-Routinen und sollten nicht in anderweitig verwendet werden, es sei denn, man ist sich über ihre Funktion im Klaren:

BYTE FUNC PF_ToLower
BYTE FUNC PF_IsDigit
CARD FUNC PF_Nbase

Der zusammengefasste Programmtext dieser Routinen befindet sich am Ende des Abschnitts.

10.1 PrintF

Zweck: Formatiertes Ausgeben von Daten.

Format: PROC PrintF(BYTE ARRAY control CARD c1,c2,c3,c4,c5,c6)

Parameter: control die Zeichenkette legt das Format der nachfolgenden Daten fest.
c1 – c6 die auszugebenden Daten.

Beschreibung: Diese Prozedur ist eine Verbesserung der PrintF-Routine aus der Action!-Library. Der Unterschied besteht in den verfügbaren Steuerungsmöglichkeiten und ihren zusätzlichen Einstellungen. Die Steuerungsmöglichkeiten selbst sind:

%D dezimale Schreibweise
%O oktale Schreibweise
%H hexadezimale Schreibweise
%U Schreibweise als CARD ohne Vorzeichen
%C Zeichen (character)
%S String (BYTE ARRAY)
%E EOL (Zeilenende)
%% '%'-Zeichen

Soweit sieht das nach der 'normalen' PrintF-Routine aus. Aber nun kommt die Verbesserung. Zwischen dem vorangestellten '%'-Zeichen und dem Steuerungszeichen (außer bei 'E' und '%') kann man folgende Informationen zur Feldgröße und der Randausrichtung einfügen:

'-' Ein Minuszeichen bewirkt eine Ausrichtung der Daten innerhalb ihres Feldes zum linken Rand (Voreinstellung ist rechts).

'n' Eine Zahl bestimmt die minimale Feldgröße für die Daten. Die Daten werden in einem Feld der mit 'n' vorgegebenen Mindestgröße ausgegeben und in einem größeren, wenn die Daten länger sind. Sind die Daten kürzer als die Feldgröße, werden sie im Feld zum rechten Rand hin ausgerichtet, es sei denn, das Minuszeichen wurde als Steuerzeichen eingesetzt.

109) Kapitel VI, Abschnitt 2.1.5.

'.' Ein Punkt gefolgt von einer Zahl bestimmt die maximale Anzahl der Zeichen, die in dem Feld ausgegeben werden.

Beispiele:
Die nachfolgende Auflistung von Strings für die Steuerung zeigt die unterschiedlichen Auswirkungen auf die Ausgabe des Strings 'Action!'. Die Ausgabe wurde jeweils in senkrechte Striche eingefasst, um die Feldgröße sichtbar werden zu lassen.

```
%S        |Action!|
%5S       |Action!|
%10S      |   Action!|
%-10S     |Action!   |
%10.4S    |      ACTI|
%-10.4S   |ACTI      |
%.4S      |ACTI|
```

10.2 PrintFD

Zweck: Formatiertes Ausgeben von Daten über einen angegebenen Kanal.

Format: PROC PrintFD(BYTE chan BYTE ARRAY control CARD c1,c2,c3,c4,c5,c6)

Parameter: chan eine Kanalnummer (0-7).
control diese Zeichenkette legt das Format der nachfolgenden Daten fest.
c1 – c6 die auszugebenden Daten.

Beschreibung: Diese Prozedur entspricht exakt der vorherigen Routine PrintF, ermöglicht aber zusätzlich die Ausgabe auf einen bestimmten Kanal und damit auf ein Gerät.

Anmerkung: Die ursprüngliche Routine funktionierte prima, solange man nicht versuchte, einen CARD-Wert größer als 32767 oder den INT-Wert '-32768' auszugeben. Der Grund lag in der PROC PF_NBase, da dort die Operatoren 'MOD' und '/' verwendet werden, die ihrerseits im Modul die Divisionsroutine aufrufen. Und diese ist eine 'signierte Division', weshalb das für große CARD-Werte nicht funktioniert. Deshalb wurde eine Routine zur 'unsignierten Division' eingefügt, die stattdessen aufgerufen wird. Außerdem erfolgte eine Änderung in der PROC PF_NBase. Daraus resultierend sollte die PrintF-Routine nun für alle CARD- und INT-Zahlen einwandfrei arbeiten.

```
;************************************
; Unsignierte Division
;************************************
  CARD Quotient, Remainder

  PROC UDiv(CARD a, divisor)
      DEFINE GETCARRY="~[$2E carry]"
      BYTE carry, i
      CARD temp
      Remainder = 0
      FOR i = 1 TO 16
        DO
```

```
          Remainder ==LSH 1
          Quotient ==LSH 1
          IF (a&$8000)#0 THEN
              Remainder ==% 1
          FI
          a ==LSH 1
          temp = Remainder - divisor
          GETCARRY
          IF (carry&1)#0 THEN
              Remainder = temp
              Quotient ==+ 1
          FI
        OD
  RETURN

;************************************
; Interne Funktion ToLower
;************************************

BYTE FUNC PF_ToLower(BYTE c)

  IF (c>='A) AND (c<='Z) THEN
    c==+32
  FI
RETURN(c)

;************************************
; Interne Funktion IsDigit
;************************************

BYTE FUNC PF_IsDigit( BYTE c )

  IF (c>='0) AND (c<='9) THEN
    RETURN(1)
  FI
RETURN(0)

;*************************************
; Interne Funktion PF_NBase
;*************************************

CARD FUNC PF_Nbase(CARD n,base  BYTE ARRAY s)

  BYTE length,ptr,d

  IF n=0 THEN
    s(1)='0
    s(0)=1
    RETURN(s)
  FI
  length=0
```

```
  ptr=17
  WHILE n>0
    DO
    UDiv( n, base )
    d=Remainder
    IF d<10 THEN
      d==+'0
    ELSE
      d==+55
    FI
    s(ptr)=d
    ptr==-1
    length==+1
    n=Quotient
    OD
  s(ptr)=length
RETURN(s+ptr)

;*************************************
; PrintF to a channel
;*************************************

PROC PrintFD(BYTE chan  BYTE ARRAY control
             CARD c1,c2,c3,c4,c5,c6)
  BYTE cptr,c,rjustify,
       zerofill,k,slen,width

  INT prcisn
  BYTE POINTER ps
  INT POINTER args
  BYTE ARRAY s(18)

  args=@c1
  cptr=1
  DO
    IF cptr>control(0) THEN
      EXIT
    FI
    c=control(cptr)
    cptr==+1
    IF c='% THEN     ; format character found
      ; check for options
      c=control(cptr)
      IF c='- THEN
        rjustify=0
        cptr==+1
        c=control(cptr)
      ELSE
        rjustify=1
      FI
      IF c='0 THEN
```

```
    zerofill=1
  ELSE
    zerofill=0
  FI
  width=0
  DO
    c=PF_ToLower(control(cptr))
    cptr==+1
    IF PF_IsDigit(c)=0 THEN
      EXIT
    FI
    width=10*width+c-'0
  OD
  IF c#'. THEN
    prcisn=32767
  ELSE
    prcisn=0
    DO
      c=PF_ToLower(control(cptr))
      cptr==+1
      IF PF_IsDigit(c)=0 THEN
        EXIT
      FI
      prcisn=10*prcisn+c-'0
    OD
  FI
  ; process conversion chars
  c=PF_ToLower(c)
  IF (c='d) OR (c='i) THEN
    IF args^<0 THEN
      ps=PF_Nbase(-args^,10,s)
      args==+2
      ps==-1
      ps(0)=ps(1)+1
      ps(1)='-
    ELSE
      ps=PF_Nbase(args^,10,s)
      args==+2
    FI
  ELSEIF c='u THEN
    ps=PF_Nbase(args^,10,s)
    args==+2
  ELSEIF (c='x) OR (c='h) THEN
    ps=PF_Nbase(args^,16,s)
    args==+2
  ELSEIF c='o THEN
    ps=PF_Nbase(args^,8,s)
    args==+2
  ELSEIF c='b THEN
    ps=PF_Nbase(args^,2,s)
    args==+2
```

```
      ELSEIF c='s THEN
        ps=args^
        args==+2
      ELSEIF c='e THEN
        s(0)=0
        ps=s
        PutDE(chan)
      ELSE
        IF c='c THEN
          c=args^
          args==+2
        FI
        s(1)=c
        s(0)=1
        ps=s
      FI
      ; now do filling and print result
      slen=ps(0)
      IF slen>prcisn THEN
        slen=prcisn
      FI
      IF rjustify=1 THEN
        WHILE width>slen
          DO
          width==-1
          IF zerofill=1 THEN
            PutD(chan,'0)
          ELSE
            PutD(chan,' )
          FI
          OD
      FI
      k=1
      WHILE (k<=ps(0)) AND (k<=prcisn)
        DO
        PutD(chan,ps(k))
        k==+1
        OD
      IF rjustify=0 THEN
        WHILE width>slen
          DO
          PutD(chan,' )
          width==-1
          OD
      FI
    ELSE     ; not a format string
    ; just put out a char
    PutD(chan,c)
    FI
  OD
RETURN
```

```
;************************************
; PrintF to default channel
;************************************

PROC PrintF(BYTE ARRAY control
            CARD c1,c2,c3,c4,c5,c6)

  PrintFD(device,control,c1,c2,c3,c4,c5,c6)
RETURN
```

11 REAL.ACT

Diese Toolkit-Datei enthält Routinen, mit denen man die Fließkommaroutinen im ROM des ATARIs von Action! aus ansprechen kann. Damit lässt sich die Sprache Action! weitaus wirkungsvoller einsetzen, wenn man Programme mit Schwerpunkt numerischer Verarbeitung schreiben will.

Damit man die Fließkommaroutinen (nachfolgend Real-Routinen genannt) nutzen kann, müssen Variablen des Typs REAL deklariert werden; z.B.:

REAL x,y,z

Der Typ REAL ist tatsächlich ein Typ Record[110], weshalb der Name der Variablen ein Zeiger auf den Datensatz selbst ist. Dadurch ist er einem Array ziemlich ähnlich.

Es ist nicht möglich per zuordnender Anweisung einen Wert an eine REAL zu übergeben, da der Action!-Compiler intern nichts mit REALs anfangen kann. Stattdessen müssen die Routinen RealAssign, ValR, IntToReal, InputR oder InputRD verwendet werden.

Zusätzlich enthält diese Toolkit-Datei einige mathematische Routinen zum Arbeiten mit REALs sowie Routinen für die Ein-/Ausgabe von REALs.

Im Anschluss an die Beschreibungen zu den Routinen finden sich einige Beispiele zu deren Verwendung. Für die Beispiele werden folgende Deklarationen angenommen:

```
REAL xreal,yreal,zreal
BYTE ARRAY astring
INT xint,yint,zint
BYTE channel
```

Die nachfolgend aufgeführten Routinen sind interne REAL-Routinen von Action! und sollten nicht anderweitig verwendet werden.

```
PROC ROM_AFP          PROC ROM_FASC
PROC ROM_IFP          PROC ROM_FPI
PROC ROM_FSUB         PROC ROM_FADD
PROC ROM_FMULT        PROC ROM_FDIV
PROC ROM_EXP          PROC ROM_EXP10
PROC ROM_LOG          PROC ROM_LOG10
PROC ROM_INIT
```

Anmerkung: In den Deklarationen der Parameter ist häufig der Typ 'REAL POINTER' zu finden. Das ist lediglich ein Hinweis darauf, dass man da den Namen (Kennung) des REALs verwenden sollte, da der Name an sich schon ein Zeiger auf das REAL ist.

Hinweis: Der komplette Programmtext der Routine REAL.ACT findet sich am Ende des Abschnitts 10.

110) Kapitel IV, Abschnitt 8.3.

11.1 Routinen zur Umwandlung von REALs

Die nachfolgenden Routinen dienen der Konvertierung von REALs.

11.1.1 IntToReal

Zweck: Übergeben eines INT-Wertes an eine REAL-Variable.

Format: PROC IntToReal(INT i REAL POINTER r)

Parameter:
- i der INT-Wert, der an die REAL übergeben wird.
- r die REAL, an die der INT-Wert übergeben wird.

Beschreibung: Diese Prozedur ermöglicht die Übergabe des Wertes einer INT an eine REAL-Variable. Wenn der Action!-Compiler REALs direkt bearbeiten könnte, entspräche das 'r=i'.

Beispiele:
```
xint=453
IntToReal(xint,xreal) ; xreal ist nun gleich 453
IntToReal(2534,yreal) ; yreal ist nun 2534
```

11.1.2 RealToInt

Zweck: den INT-Wert einer REAL-Variablen ausgeben.

Format: INT FUNC RealToInt(REAL POINTER r)

Parameter:
- r die REAL-Variable.

Beschreibung: Diese Funktion gibt den INT-Wert einer REAL aus, der als Parameter an sie übergeben wurde.

Beispiel:
```
xint=RealToInt(xreal); xint ist gleich dem INT-Wert von xreal
```

11.1.3 StrR (RealToString)

Zweck: Eine REAL in einen String umwandeln.

Format: PROC StrR(REAL POINTER r BYTE ARRAY s)

Parameter:
- r die REAL, die umgewandelt wird.
- s der String, in dem die Zeichen gespeichert werden, welche der Real entsprechen.

Beschreibung: Diese Prozedur wandelt eine REAL in die ihr entsprechenden Zeichen um.

Beispiele:
```
IntToReal(3926,xreal) ; xreal = 3926
StrR(xreal,astring) ; astring enthält nun '3926'
```

11.1.4 ValR (StringToReal)

Zweck: Einen String in eine REAL umwandeln.

Format: PROC ValR(BYTE ARRAY s REAL POINTER r)

Parameter:
- s der String, der umgewandelt wird.
- r die REAL, welcher der Wert von 's' übergeben wird.

Beschreibung: Diese Prozedur wandelt soviel wie möglich aus einem String in eine REAL-Variable um. Allerdings liefert ein String wie 'abcde' keine Zahlen und die Routine setzt daher den Wert 0 in die REAL.

Beispiele: astring="45.276"
ValR(astring,xreal) ; entspricht xreal=45.276
ValR("2.7E-4",yreal) ; entspricht yreal=$2.7*10^{-4}$
ValR("70.2agr",zreal) ; entspricht zreal=70.2

11.2 Mathematikroutinen für REALs

Dieser Abschnitt befasst sich mit Routinen zum Rechnen mit REALs.

11.2.1 RealAssign

Zweck: Den Wert einer REAL-Variablen einer anderen zuweisen.

Format: PROC RealAssign(REAL POINTER a,b)

Parameter: a der REAL-Wert, der übergeben wird.
b die REAL, welcher der Wert zugewiesen wird.

Beschreibung: Mit dieser Prozedur kann man den Wert einer REAL einer anderen zuweisen. Sofern der Action!-Compiler REALs verarbeiten könnte, entspräche das 'b=a'.

Beispiele: RealAssign(xreal,yreal) ; entspricht yreal=xreal
RealAssign(zreal,yreal) ; entspricht zreal=yreal

11.2.2 RealAdd

Zweck: Zwei REALs addieren.

Format: PROC RealAdd(REAL POINTER a,b,c)

Parameter: a ein Summand
b ein Summand
c die Summe

Beschreibung: Mit dieser Prozedur kann man zwei REALs addieren. Sofern der Action!-Compiler REALs verarbeiten könnte, entspräche das 'c=a+b'.

Beispiel: RealAdd(xreal,yreal,zreal) ; entspricht zreal=xreal+yreal

11.2.3 RealSub

Zweck: Zwei REALs subtrahieren.

Format: PROC RealSub(REAL POINTER a,b,c)

Parameter: a der Subtrahend
b der Minuend
c die Differenz

Beschreibung: Mit dieser Prozedur kann man zwei REALs voneinander subtrahieren. Sofern der Action!-Compiler REALss verarbeiten könnte, entspräche das 'c=a-b'.

Beispiel: RealSub(xreal,yreal,zreal) ; entspricht zreal=xreal-yreal

11.2.4 RealMult

Zweck: Zwei REALs multiplizieren.

Format: PROC RealMult(REAL POINTER a,b,c)

Parameter:
- a der Multiplikant
- b der Multiplikator
- c das Produkt

Beschreibung: Mit dieser Prozedur kann man zwei REALs miteinander multiplizieren. Sofern der Action!-Compiler REALs verarbeiten könnte, entspräche das 'c=a*b'.

Beispiel: RealSub(xreal,yreal,zreal) ; entspricht zreal=xreal*yreal

11.2.5 RealDiv

Zweck: Zwei REALs dividieren.

Format: PROC RealDiv(REAL POINTER a,b,c)

Parameter:
- a der Dividend
- b der Divisor
- c der Quotient

Beschreibung: Mit dieser Prozedur kann man zwei REALs durcheinander dividieren. Sofern der Action!-Compiler REALs verarbeiten könnte, entspräche das 'c=a/b'.

Beispiel: RealDiv(xreal,yreal,zreal) ; entspricht zreal=xreal/yreal

11.2.6 Exp

Zweck: Die Zahl e zur Potenz a erheben.

Format: PROC Exp(REAL POINTER a,b)

Parameter:
- a die Potenz, zu der *e* erhoben wird.
- b das Ergebnis der Potenzierung von *e* mit a.

Beschreibung: Mit dieser Prozedur kann man den Exponenten zur Basis *e* für eine REAL bestimmen. Das entspricht dem Ausdruck 'b=e^a

Beispiel: Exp(xreal,yreal) ; entspricht yreal=e^{xreal}

11.2.7 Exp10

Zweck: Die Zahl 10 zur Potenz a erheben.

Format: PROC Exp10(REAL POINTER a,b)

Parameter:
- a die Potenz, zu der 10 erhoben wird.
- b das Ergebnis der Potenzierung von 10 mit a.

Beschreibung: Mit dieser Prozedur kann man den Exponenten zur Basis 10 für eine REAL bestimmen. Das entspricht dem Ausdruck 'b=10^a

Beispiel: Exp10(xreal,yreal) ; entspricht yreal=10^{xreal}

11.2.8 Power

Zweck: Eine REAL zur Potenz REAL erheben.

Format: PROC Power(REAL POINTER a,b,c)

Parameter:
- a die Basis zur Potenz.
- b die Potenz, zu der a erhoben wird.
- c das Ergebnis der Potenzierung von a mit b.

Beschreibung: Mit dieser Prozedur kann man eine REAL zu einer Potenz erheben, die von einer anderen REAL bestimmt wird. Das entspricht dem Ausdruck 'c=a^b

Beispiel: Power(xreal,yreal,zreal) ; entspricht zreal=$xreal^{yreal}$

11.2.9 Ln (natürlicher Logarithmus)

Zweck: Den natürlichen Logarithmus einer REAL bestimmen.

Format: PROC Ln(REAL POINTER a,b)

Parameter:
- a die REAL, deren natürlicher Logarithmus bestimmt wird.
- b das Ergebnis der Bestimmung.

Beschreibung: Mit dieser Prozedur kann man den natürlichen Logarithmus (Basis *e*) einer REAL bestimmen. Das entspricht dem Ausdruck 'b=ln(a)'.

Beispiel: Ln(xreal,yreal) ; entspricht yreal=ln(xreal)

11.2.10 Log10

Zweck: Den normalen Logarithmus einer REAL bestimmen.

Format: PROC Log10(REAL POINTER a,b)

Parameter:
- a die REAL, deren Logarithmus bestimmt wird.
- b das Ergebnis der Bestimmung.

Beschreibung: Mit dieser Prozedur kann man den normalen Logarithmus (Basis 10) einer REAL bestimmen. Das entspricht dem Ausdruck 'b=log(a)'.

Beispiel: Log10(xreal,yreal) ; entspricht yreal=log(xreal)

11.3 Ein-/Ausgabe-Routinen

I/O-Routinen für die Verarbeitung mit REALs.

11.3.1 PrintR

Zweck: Ausgabe einer REAL an das aktuelle Gerät.

Format: PROC PrintR(REAL POINTER a)

Parameter: a die auszugebende REAL.

Beschreibung: Mit dieser Prozedur kann man eine REAL-Zahl auf das aktuelle Gerät ausgeben ohne ein EOL-Zeichen (RETURN) am Ende.

11.3.2 PrintRD

Zweck: Ausgabe einer REAL über einen bestimmten Kanal.

Format: PROC PrintRD(BYTE channel REAL POINTER a)

Parameter: channel der Kanal für die Ausgabe.
a die auszugebende REAL.

Beschreibung: Mit dieser Prozedur kann man eine REAL-Zahl auf das durch den Kanal bestimmte Gerät ausgeben ohne ein EOL-Zeichen (RETURN) am Ende.

11.3.3 PrintRE

Zweck: Ausgabe einer REAL an das aktuelle Gerät mit EOL.

Format: PROC PrintRE(REAL POINTER a)

Parameter: a die auszugebende REAL.

Beschreibung: Mit dieser Prozedur kann man eine REAL-Zahl auf das aktuelle Gerät ausgeben mit einem EOL-Zeichen (RETURN) am Ende.

11.3.4 PrintRDE

Zweck: Ausgabe einer REAL über einen bestimmten Kanal mit EOL.

Format: PROC PrintRDE(BYTE channel REAL POINTER a)

Parameter: channel der Kanal für die Ausgabe.
a die auszugebende REAL.

Beschreibung: Mit dieser Prozedur kann man eine REAL-Zahl auf das durch den Kanal bestimmte Gerät ausgeben mit einem EOL-Zeichen (RETURN) am Ende.

11.3.5 InputR

Zweck: Eingabe einer REAL vom aktuellen Gerät.

Format: PROC InputR(REAL POINTER a)

Parameter: a die REAL, in welcher der eingegebene Wert gespeichert wird.

Beschreibung: Mit dieser Prozedur kann man eine REAL-Zahl vom aktuellen Gerät einlesen und in der angegebenen REAL speichern.

11.3.6 InputRD

Zweck: Eingabe einer REAL über einen bestimmten Kanal.

Format: PROC InputRD(BYTE channel REAL POINTER a)

Parameter: channel der Kanal für die Eingabe.
a die REAL, in welcher der eingegebene Wert gespeichert wird.

Beschreibung: Mit dieser Prozedur kann man eine REAL-Zahl von dem per Kanal bestimmten Gerät einlesen und in der angegebenen REAL speichern.

11.4 Programmtext

```
MODULE

TYPE REAL=[CARD r1,r2,r3] ; Das ATARI Format fuer Fliesskomma-
                          ; zahlen ist 6 Bytes lang.

BYTE Cix=$F2

INT  Fr0Int=$D4

CARD InBuff=$F3

REAL Fr0=$D4, Fr1=$E0

BYTE ARRAY LBuff=$580

;*************************************
; interne Fliesskommaroutinen
;*************************************

PROC ROM_AFP=$D800()
PROC ROM_FASC=$D8E6()
PROC ROM_IFP=$D9AA()
PROC ROM_FPI=$D9D2()
PROC ROM_FSUB=$DA60()
PROC ROM_FADD=$DA66()
PROC ROM_FMULT=$DADB()
PROC ROM_FDIV=$DB28()
PROC ROM_EXP=$DDC0()
PROC ROM_EXP10=$DDCC()
PROC ROM_LOG=$DECD()
PROC ROM_LOG10=$DED1()
PROC ROM_INIT=$DA51()

;**************************************
; Eine Dummy-Prozedur fuer den Zugriff
; auf die Fliesskommaroutinen
;**************************************
```

```
PROC Junk()
RETURN

;************************************
; REAL einer anderen REAL zuweisen
;************************************

PROC RealAssign(REAL POINTER a,b)

  b.r1=a.r1
  b.r2=a.r2
  b.r3=a.r3
RETURN

;************************************
; INT in REAL umwandeln
;************************************

PROC IntToReal(INT i  REAL POINTER r)

  Fr0Int=i
  Junk=ROM_IFP
  Junk()
  RealAssign(Fr0,r)
RETURN

;************************************
; REAL in INT umwandeln
;************************************

INT FUNC RealToInt(REAL POINTER r)

  RealAssign(r,Fr0)
  ROM_FPI()
RETURN(Fr0Int)

;************************************
; REALs subtrahieren
;************************************

PROC RealSub(REAL POINTER a,b,c)

  RealAssign(a,Fr0)
  RealAssign(b,Fr1)
  ROM_FSUB()
  RealAssign(Fr0,c)
RETURN
;
;
;
;
```

```
;************************************
; REALs addieren
;************************************

PROC RealAdd(REAL POINTER a,b,c)

  RealAssign(a,Fr0)
  RealAssign(b,Fr1)
  ROM_FADD()
  RealAssign(Fr0,c)
RETURN

;************************************
; REALs multiplizieren
;************************************

PROC RealMult(REAL POINTER a,b,c)

  RealAssign(a,Fr0)
  RealAssign(b,Fr1)
  ROM_FMULT()
  RealAssign(Fr0,c)
RETURN

;************************************
; REALs dividieren
;************************************

PROC RealDiv(REAL POINTER a,b,c)

  RealAssign(a,Fr0)
  RealAssign(b,Fr1)
  ROM_FDIV()
  RealAssign(Fr0,c)
RETURN

;************************************
; REAL in ASCII String umwandeln
;************************************

PROC StrR(REAL POINTER r  BYTE ARRAY s)

  BYTE i,c

  BYTE POINTER ptr

  RealAssign(r,Fr0)
  ROM_FASC()
  ptr=InBuff
  WHILE ptr^='0
    DO
```

```
    ptr==+1
    OD
  i=0
  DO
    c=ptr(i)
    i==+1
    s(i)=c&$7F
  UNTIL c&$80
  OD
  s(0)=i
RETURN

;************************************
; String in REAL umwandeln
;************************************

PROC ValR(BYTE ARRAY s  REAL POINTER r)

  BYTE i

  FOR i=1 TO s(0)
    DO
    LBuff(i-1)=s(i)
    OD
  LBuff(i-1)=0   ; AN INVALID VALUE
  InBuff=LBuff
  Cix=0
  ROM_AFP()
  RealAssign(Fr0,r)
RETURN

;************************************
; Base E Potenzierung
;************************************

PROC Exp(REAL POINTER a,b)

  RealAssign(a,Fr0)
  ROM_EXP()
  RealAssign(Fr0,b)
RETURN

;************************************
; Base 10 Potenzierung
;************************************

PROC Exp10(REAL POINTER a,b)

  RealAssign(a,Fr0)
  ROM_EXP10()
  RealAssign(Fr0,b)
```

```
RETURN

;************************************
; Natuerlicher Logarithmus
;************************************

PROC Ln(REAL POINTER a,b )

  RealAssign(a,Fr0)
  ROM_LOG()
  RealAssign(Fr0,b)
RETURN

;************************************
; Logarithmus zur Basis 10
;************************************

PROC Log10(REAL POINTER a,b)

  RealAssign(a,Fr0)
  ROM_LOG10()
  RealAssign(Fr0,b)
RETURN

;************************************
; Potenzierung
;************************************

PROC Power(REAL POINTER a,b,c)

  Ln(a,c)
  RealMult(b,c,c)
  Exp(c,c)
RETURN

;************************************
; REAL auf bestimmtes Geraet ausgeben
;************************************

PROC PrintRD(BYTE d  REAL POINTER a)

  BYTE ARRAY temp(20)

  StrR(a,temp)
  PrintD(d,temp)
RETURN

;************************************
; REAL auf aktuelles Geraet ausgeben
;************************************
```

```
PROC PrintR(REAL POINTER a)

  PrintRD(device,a)
RETURN

;************************************
; REAL auf bestimmtes Geraet ausgeben
: mit EOL am Ende
;************************************

PROC PrintRDE(BYTE d  REAL POINTER a)

  PrintRD(d,a)
  PutDE(d)
RETURN

;************************************
; REAL auf aktuelles Geraet ausgeben
: mit EOL am Ende
;************************************

PROC PrintRE(REAL POINTER a)

  PrintRDE(device,a)
RETURN

;************************************
; REAL von einem bestimmten Geraet
: einlesen
;************************************

PROC InputRD(BYTE d  REAL POINTER a)

  BYTE ARRAY temp(128)

  InputMD(d,temp,126)
  ValR(temp,a)
RETURN

;************************************
; REAL vom aktuellen Geraet einlesen
;************************************

PROC InputR(REAL POINTER a)

  InputRD(device,a)
RETURN
```

12 SORT.ACT

Diese vier Sortierroutinen verwenden alle den Quick-Sort-Algorithmus. Der Algorithmus wurde wegen seiner Schnelligkeit gewählt. Im günstigsten Fall (bei nicht vorsortierten Daten und mittig gewähltem Pivotelement) ist Quick-Sort einer der schnellsten bekannten Sortieralgorithmen. Andere Sortier-Algorithmen können da kaum mithalten.

Bei genauer Betrachtung des Programmtextes von SORT.ACT wird deutlich, dass man eigene Routinen zum Sortieren von REALs oder komplexen Record-Typen einfach dadurch entwickeln kann, dass man eigene Vergleichs- und Austauschroutinen schreibt.

Anmerkung zur Anwendung: Bevor man eine der Sortierroutinen einsetzt, sollte man erst die Programmzeile

```
DEFINE SortMax="10000"
```

auf die maximale Größe des geplanten Datenarrays setzen. Eine Alternative wäre die Änderung der Sortierroutine dergestalt, dass in sie mit INCLUDE die Routinensammlung ALLOC.ACT eingefügt wird und dann dynamisch 'List'-Arrays erzeugt werden.

Hinweis: Der Programmtext der kompletten Routine SORT.ACT findet sich am Ende von Abschnitt 11.

12.1 SortB

Zweck: Ein-Byte-Daten aufsteigend oder absteigend sortieren.

Format: PROC SortB(BYTE ARRAY data CARD len BYTE order)

Parameter:
- data — Das Array, das die zu sortierenden Daten enthält.
- len — Die Länge des Datenarrays.
- order — Legt die Sortierfolge fest (0=auf-, 1=absteigend).

Beschreibung: Sehr schnelle Sortierung von Ein-Byte-Daten.

12.2 SortC

Zweck: Zwei-Byte-Daten ohne Vorzeichen (unsigned) aufsteigend oder absteigend sortieren.

Format: PROC SortC(CARD ARRAY data CARD len BYTE order)

Parameter:
- data — Das Array, das die zu sortierenden Daten enthält.
- len — Die Länge des Datenarrays.
- order — Legt die Sortierfolge fest (0=auf-, 1=absteigend).

Beschreibung: Sehr schnelle Sortierung von Zwei-Byte-Daten.

12.3 SortI

Zweck: Zwei-Byte-Daten mit Vorzeichen (signed) aufsteigend oder absteigend sortieren.

Format: PROC SortI(INT ARRAY data CARD len BYTE order)

Parameter: data Das Array, das die zu sortierenden Daten enthält.
len Die Länge des Datenarrays.
order Legt die Sortierfolge fest (0=auf-, 1=absteigend).

Beschreibung: Sehr schnelle Sortierung von Zwei-Byte-Daten mit Vorzeichen (signed).

12.4 SortS

Zweck: String-Daten aufsteigend oder absteigend sortieren.

Format: PROC SortS(CARD ARRAY data CARD len BYTE order)

Parameter: data Das Array, das die Adressen der zu sortierenden Strings enthält.
len Die Länge des Datenarrays.
order Legt die Sortierfolge fest (0=auf-, 1=absteigend).

Beschreibung: Sehr schnelle Sortierung von Strings. Darauf achten, dass die Adressen der zu sortierenden Strings die Elemente des CARD ARRAY sein müssen.

12.5 Programmtext

```
MODULE

; Im Gegensatz zur Anmerkung am Anfang des Abschnitts 11
; wird die Definition von SortMax von den Sortierroutinen
; ignoriert und muss nicht geaendert werden auf den
; geplanten Datenumfang. Dies wurde durch eine Verbesserung
; der Arbeitsweise der Sortierroutinen erreicht.
;

DEFINE SortMax="10000"

CARD ListSize

BYTE ARRAY BArray

INT ARRAY IArray

CARD ARRAY CArray,
           List(64) ; groß genug fuer 64K Elemente


;************************************
; BYTE Vergleich
;************************************

BYTE FUNC BDescend(CARD i,j)

  IF BArray(i)>BArray(j) THEN
    RETURN (1)
  FI
RETURN (0)
```

```
;************************************

BYTE FUNC BAscend(CARD i,j)

  IF BArray(i)<BArray(j) THEN
    RETURN (1)
  FI
RETURN (0)

;************************************
; CARD Vergleich
;************************************

BYTE FUNC CDescend(CARD i,j)

  IF CArray(i)>CArray(j) THEN
    RETURN (1)
  FI
RETURN (0)

;************************************

BYTE FUNC CAscend(CARD i,j)

  IF CArray(i)<CArray(j) THEN
    RETURN (1)
  FI
RETURN (0)

;************************************
; INT Vergleich
;************************************

BYTE FUNC IDescend(CARD i,j)

  IF IArray(i)>IArray(j) THEN
    RETURN (1)
  FI
RETURN (0)

;************************************
; String Vergleich
;************************************

BYTE FUNC IAscend(CARD i,j)

  IF IArray(i)<IArray(j) THEN
    RETURN (1)
  FI
RETURN (0)
```

```
;**********************************

BYTE FUNC SDescend(CARD i,j)

  IF SCompare(CArray(i),CArray(j))>0 THEN
    RETURN (1)
  FI
RETURN (0)

;**********************************

BYTE FUNC SAscend(CARD i,j)

  IF SCompare(CArray(i),CArray(j))<0 THEN
    RETURN (1)
  FI
RETURN (0)

;**********************************
; Tausch von 2 BYTE Elementen
;**********************************

PROC BSwap(CARD i,j)

  BYTE temp

  temp=BArray(i)
  BArray(i)=BArray(j)
  BArray(j)=temp
RETURN

;****************************************
; Tausch von 2 Elementen – CARD oder String
;****************************************

PROC CSwap(CARD i,j)

  CARD temp

  temp=CArray(i)
  CArray(i)=CArray(j)
  CArray(j)=temp
RETURN

;**********************************
; Tausch von 2 INT Elementen
;**********************************

PROC ISwap(CARD i,j)

  INT temp
```

```
  temp=IArray(i)
  IArray(i)=IArray(j)
  IArray(j)=temp
RETURN

;*************************************
; Die naechsten beiden Routinen zeigen
; auf eine der obigen Vergleichs- und
; Tauschroutinen, abhaengig vom sor-
; tierten Datentyp und der Reihenfolge
; der Sortierung.
;*************************************

BYTE FUNC Compare(CARD i,j)

PROC Swap(CARD i,j)

;************************************
; Einen Bereich der Liste hinzufuegen
;************************************

PROC AddList(CARD low,high)

  IF high+1>low+1 THEN
    List(ListSize)=low
    ListSize==+1
    List(ListSize)=high
    ListSize==+1
  FI
RETURN

;************************************
; Auslesen des letzten low,high Paars
; von der Bereichsliste
;************************************

PROC GetFirst(CARD POINTER lowP,highP)

  ListSize==-1
  highP^=List(ListSize)
  ListSize==-1
  lowP^=List(ListSize)
RETURN

;************************************
; Sortierarray in Bereiche aufteilen
;************************************

CARD FUNC Partition(CARD low,high)

  CARD i,j,pivot,mid
```

```
  ; Find median of 1st,middle,and last
  ; elements to use as pivot for partitioning.
  mid=(low+high) RSH 1
  IF Compare(mid,low) THEN
    Swap(low,mid)
  FI
  IF Compare(high,low) THEN
    Swap(low,high)
  FI
  IF Compare(mid,high) THEN
    Swap(mid,high)
  FI
  pivot=high
  i=low
  j=high
  WHILE i<j
    DO
    WHILE Compare(i,pivot)
      DO
      i==+1
      OD
    WHILE (Compare(j,pivot)=0) AND (j>i)
      DO
      j==-1
      OD
    IF i<j THEN
      Swap(i,j)
    FI
    OD
  Swap(i,high)
RETURN (i)

;************************************
; Sortieren per QuickSort-Algorithmus
;************************************

PROC QuickSort(CARD len)

  CARD low,middle,high,i

  ListSize=0
  AddList(0,len-1)
  WHILE ListSize>0
    DO
    GetFirst(@low,@high)
    middle=Partition(low,high)
    ; Put larger partition onto stack first
    ; in order to decrease maximum stack size.
    IF (middle-low) > (high-middle) THEN
      AddList(low,middle-1)
      AddList(middle+1,high)
```

```
    ELSE
      AddList(middle+1,high)
      AddList(low,middle-1)
    FI
    OD
RETURN

;************************************
; Ein BYTE ARRAY sortieren
;************************************

PROC SortB(BYTE ARRAY data CARD len BYTE order)

  IF order THEN
    Compare=BDescend
  ELSE
    Compare=BAscend
  FI
  Swap=BSwap
  BArray=data
  QuickSort(len)
RETURN

;************************************
; Ein CARD ARRAY sortieren
;************************************

PROC SortC(CARD ARRAY data CARD len BYTE order)

  IF order THEN
    Compare=CDescend
  ELSE
    Compare=CAscend
  FI
  Swap=CSwap
  CArray=data
  QuickSort(len)
RETURN

;************************************
; Ein INT ARRAY sortieren
;************************************

PROC SortI(INT ARRAY data CARD len BYTE order)

  IF order THEN
    Compare=IDescend
  ELSE
    Compare=IAscend
  FI
  Swap=ISwap
```

```
  IArray=data
  QuickSort(len)
RETURN

;*************************************
; Sortieren eines CARD ARRAY, dessen
; Elemente die Adressen der Strings sind
;*************************************

PROC SortS(CARD ARRAY data CARD len BYTE order)

  IF order THEN
    Compare=SDescend
  ELSE
    Compare=SAscend
  FI
  Swap=CSwap
  CArray=data
  QuickSort(len)
RETURN
```

13 TURTLE.ACT

Die Routinen in dieser Datei implementieren eine Turtle-Grafik ähnlich der von LOGO. Voraussetzung dafür ist ein Grafikmodus, in dem Plot und DrawTo verwendet werden können. Außerdem, da die Länge einer zu zeichnenden Linie vom Grafikmodus abhängig ist, wird keine Prüfung der Bildbegrenzung durchgeführt.

Ferner hängt die Farbe einer gezeichneten Linie ausschließlich vom momentanen Wert der Systemvariablen Color ab, weshalb man SetColor[111] zur Wahl der gewünschten Farbe verwenden sollte.

Die beiden Routinen CARD FUNC TG_ISin
CARD FUNC TG_ICos

sind interne Routinen der Turtle-Grafik und sollten nicht anderweitig verwendet werden.

Hinweis: Der komplette Programmtext der Routine TURTLE.ACT findet sich am Ende des Abschnitts 12.

13.1 Right

Zweck: Turtle um 'theta' Grad nach rechts drehen.

Format: PROC Right(INT theta)

Parameter: theta Winkel zum Drehen der Turtle im Uhrzeigersinn.

Beschreibung: Mit dieser Routine kann man die Turtle im Uhrzeigersinn um eine bestimmte Gradzahl drehen.

13.2 Left

Zweck: Turtle um 'theta' Grad nach links drehen.

Format: PROC Left(INT theta)

Parameter: theta Drehwinkel der Turtle gegen den Uhrzeigersinn.

Beschreibung: Mit dieser Routine kann man die Turtle gegen den Uhrzeigersinn um eine bestimmte Gradzahl drehen.

13.3 Turn

Zweck: Turtle links oder rechts herum drehen.

Format: PROC Turn(INT theta)

Parameter: theta Drehwinkel der Turtle.
Beschreibung: Mit dieser Routine kann man die Turtle entweder mit oder gegen den Uhrzeigersinn drehen in Abhängigkeit vom Vorzeichen für die Winkelangabe 'theta'. Ist 'theta' positiv, wird die Turtle nach rechts gedreht, andernfalls nach links.

111) Kapitel VI, Abschnitt 5.2.

13.4 Forward

Zweck: Turtle um eine bestimmte Strecke vorwärts bewegen.

Format: PROC Forward(INT length)

Parameter: length Strecke für die Vorwärtsbewegung.

Beschreibung: Mit dieser Routine kann man die Turtle um eine bestimmte Strecke vorwärts bewegen. Die Länge der Strecke hängt ab vom gewählten Grafikmodus.

13.5 SetTurtle

Zweck: Turtle an ein bestimmte x/y-Position setzen in einem bestimmten Winkel.

Format: PROC SetTurtle(INT x,y,theta)

Parameter: x horizontale Position für das Setzen der Turtle.
y vertikale Position für das Setzen der Turtle.
theta Drehwinkel, in dem die Turtle gesetzt wird.

Beschreibung: Mit dieser Prozedur kann man die Turtle an eine absolute x/y-Position setzen und in eine vorgegebene Richtung zeigen lassen. Mit Drehwinkel 'theta' zeigt die Turtle bei 0 Grad nach rechts, bei 90 Grad nach oben, bei 180 Grad nach links und bei 270 Grad nach unten. Anders ausgedrückt dreht sich die Turtle mit größer werdendem Winkel nach rechts, mit kleiner Werdendem nach links.

13.6 Programmtext

```
MODULE

INT TG_Phi
CARD TG_CurX, TG_CurY

CARD ARRAY TG_SinTab(91)=[
    0     2     4     7     9
    11    13    16    18    20
    22    24    27    29    31
    33    35    37    40    42
    44    46    48    50    52
    54    56    58    60    62
    64    66    68    70    72
    73    75    77    79    81
    82    84    86    87    89
    91    92    94    95    97
    98    99    101   102   104
    105   106   107   109   110
    111   112   113   114   115
    116   117   118   119   119
    120   121   122   122   123
    124   124   125   125   126
    126   126   127   127   127
    128   128   128   128   128
    128]
```

```
;************************************
; Turtle drehen
;************************************

PROC Turn(INT theta)

  TG_Phi=TG_Phi+theta
  WHILE TG_Phi<0
    DO
    TG_Phi==+360
    OD
  TG_Phi==MOD 360
RETURN

;************************************
; Turtle im Uhrzeigersinn drehen
;************************************

PROC Right(INT theta)

  Turn(-theta)
RETURN

;**************************************
; Turtle gegen den Uhrzeigersinn drehen
;**************************************

PROC Left(INT theta)

  Turn(theta)
RETURN

;************************************
; Interne Sinusfunktion
;************************************

CARD FUNC TG_ISin(CARD theta)

  INT sign

  sign=1
  IF theta>180 THEN
    theta=360-theta
    sign=-1
  FI
  IF theta>90 THEN
    theta=180-theta
  FI
RETURN(sign*TG_SinTab(theta))
```

```
;************************************
; Interne Cosinusfunktion
;************************************

CARD FUNC TG_ICos(CARD theta)

  INT sign

  sign=1
  IF theta>180 THEN
    theta=360-theta
  FI
  IF theta>90 THEN
    theta=180-theta
    sign=-1
  FI
RETURN(sign*TG_SinTab(90-theta))

;**************************************
; Turtle um eine bestimmte Anzahl an
; 'Streckeneinheiten' vorwaerts bewegen
;**************************************

PROC Forward(INT length)

  INT deltaX, deltaY

  deltaX=length*TG_ICos(TG_Phi)
  deltaY=length*TG_ISin(TG_Phi)
  TG_CurX==+deltaX
  TG_CurY==-deltaY
  Drawto(TG_CurX RSH 7,TG_CurY RSH 7)
RETURN

;************************************
; Turtle an (x,y) setzen und in die
; Richtung 'theta' zeigen lassen.
;************************************

PROC SetTurtle(INT x,y,theta)

  BYTE temp

  TG_CurX=x LSH 7
  TG_CurY=y LSH 7
  TG_Phi=theta
  temp=color
  color=0
  Plot(x,y)
  color=temp
RETURN
```

14 GEM.DEM

Das Spiel wurde von Joel Gluck geschrieben, nachdem er das Action!-Modul erst zwei Tage besaß. Beim Lesen des Codes fällt sofort die Ähnlichkeit zu BASIC auf. Das spiegelt Joels Programmiererfahrung (ausschließlich BASIC) wider und rührt nicht daher, dass GEM ursprünglich in BASIC geschrieben worden wäre (was nicht der Fall ist). Soviel zur Geschichte von GEM. Das Spiel ist für 1 bis 4 Spieler mit Joystick ausgelegt. Ziel ist den Edelstein in der Mitte des Bildschirms zu greifen und damit in die eigene Basis zurückzukehren, bevor man von einem der Roboter oder einem anderen Spieler abgeballert wird. Bevor das eigentliche Spiel beginnt, wird man noch gefragt:

How many points wins?
(Bei wie vielen Punkten gewinnt man?)

How many robots in final round?
(Wie viele Roboter in der Schlussrunde?)

Punkte: Einer für jeden in die eigene Ecke gebrachten Edelstein.

Ballern: Roboter oder andere Spieler kann man abballern, indem der Feuerknopf am Joystick gedrückt wird, während man den Joystick in Richtung auf das Ziel bewegt.

Getroffen: Wird man getroffen, wird die Spielfigur in der eigenen Ecke neu eingesetzt. Ein ggf. mitgeführter Edelstein ist verloren. Die Anzahl der 'Leben' ist unendlich.

Sieg: Wer die vorgegebene Punktzahl erreicht gewinnt. Man kann dann ein Neues beginnen oder beenden und in den Monitor zurückkehren durch Beantworten der Frage

Play again (y/n)?
(Noch einmal spielen (j/n)?)

Anmerkungen:
Das Spiel erfordert viel Speicherplatz und kann nicht unter allen DOS in den Editor geladen und kompiliert werden. Stattdessen RUN verwenden.

Eine zu hohe Zahl an Robotern kann Fehler verursachen. Die Höchstzahl bisher überlebter Roboter beträgt als Einzelspieler 45.

15 KALSCOPE.DEM

Diese Demonstration verwendet fortgeschrittene Algorithmen zur Berechnung und für die Display List[112], um die Kaleidoskopeffekte auf den Bildschirm zu bringen. Die Geschwindigkeit ist überraschend hoch. Man kann sogar Tempo und Persistenz (Verweildauer der Bildpunkte auf dem Bildschirm) per Joystick in Port 1 durch vertikale und horizontale Bewegung verändern. Mit gedrückt gehaltenem Feuerknopf wird die Ausgabe angehalten. Beschäftigt man sich eine Weile damit, wird man über Anzahl und Vielfalt der unterschiedlichen Muster erstaunt sein.

16 MUSIC.DEM

In diesem Programm werden einige Routinen aus dem Toolkit und das Wissen um die Tastaturmatrix des ATARI zur Simulation einer Orgel genutzt, die auf Tastendruck spielt.

112) ABBUC-Edition des ATARI Profibuchs.

Auf dem Bildschirm wird eine Klaviatur über drei Oktaven dargestellt. Die Buchstaben auf den Tasten stehen für die Noten in internationaler Notation. Die Buchstaben über und unter der Klaviatur sind die Tasten des ATARI, die man drücken muss, um den entsprechenden Ton zu hören. Die mittlere Oktave wird mit <SHIFT><Note> gespielt, die hohe Oktave mit <CONTROL><Note>.

Die Orgel ist insofern auf dem ATARI ungewöhnlich, als die Noten nur so lange ertönen, wie die Taste gedrückt wird. Die meisten Anwender wissen nicht, wie man die Dauer eines Tastendrucks bestimmen kann. Daher kann das Studium dieses Quellcodes interessante Details aufzeigen.

17 SNAILS.DEM

Ein vertrautes Spielprinzip bekannt von TRON, SNAKE, SNAIL und ähnlichen Spielen. Ausgelegt für zwei Spieler mit Joystick. Gewinner ist, wer zuerst 10 Punkte erzielt hat. Verlieren beide 'Snails' gleichzeitig 'ihr Leben', bekommt kein Spieler einen Punkt.

Dieses Spiel wurde in BASIC XL geschrieben und nach Action! konvertiert, was beim Studium des Quellcodes deutlich wird. Die ursprüngliche Version in BASIC XL findet sich auf der Disk des 'BASIC XL Toolkit'.

18 WARP.DEM

'Warp Attack' ist ein Weltraumballerspiel, in dem man 'Fliegende Weltraumfestungen' abschießen muss, die sich mit Plasmawirbeln verteidigen. Wird man getroffen, explodiert der Jäger. Zuerst muss das linke, dann das rechte Triebwerk des Gegners weggeschossen werden, dann kann man den Rest zerstören.

Das Spiel wurde mit fortschrittlichen Programmiertechniken realisiert. Darunter sind eine manipulierte Display List, Display List Interrupts, Vertical Blank Interrupts sowie eine schnelle Blockroutine zum 'Zeichnen'. DLI und VBI[113] werden gemeinsam zum Erzeugen der scrollenden Planetenoberfläche verwendet, die schnelle Blockroutine dient zum Bewegen der 'Fliegenden Weltraumfestung', die kein Player ist.

Die Datei kann nur gestartet werden, wenn sie von Disk kompiliert wurde, da sie schlicht zu groß ist. Action! kann nicht gleichzeitig Sourcecode und Objektcode im Speicher halten. Mit DOS XL und SpartaDOS X ist es aber möglich.

113) ABBUC-Edition des ATARI Profibuchs.

Anhang

A - Sprachsyntax von Action!

Allgemeines

Nachfolgend die Syntax von Action! in Backus-Naur-Form. Diese Form verwendet einige spezielle Symbole:

Symbol	Bedeutung
::=	ist definiert als
		oder
{}	optional

A.1 Action! Konstante

Numerische Konstante
<num const> ::= <dec num> | <hex num> | <char>
<dec num> ::= <dec num><digit> | <digit>
<hex num> ::= <hexnum><hex digit> | $<hex digit>
<char> ::= '<any printable character>
<hex digit> ::= <digit> | A | B | C | D | E | F
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

String Konstante
<str const> ::= "<string>"
<string> ::= <string><str char> | <str char>
<str char> ::= <all printable characters, except " >

Compiler Konstante
<comp const> ::= <comp const>+<base comp const> | <base comp const>
<base comp const> ::= <identifier> | <num const> | <ptr ref> | *

A.2 Operatoren und elementare Datentypen

Operatoren
<special op> ::= AND | OR | & | %
<rel op> ::= XOR | ! | = | # | <> | < | <= | > | >=
<add op> ::= + | -
<mult op> ::= * | / | MOD | LSH | RSH
<unary op> ::= @ | -

Elementare Datentypen
<fund type> ::= CARD | CHAR | BYTE | INT

A.3 Action! Programmstruktur

Action!-Programm
<program> ::= <program> MODULE <prog module> | {MODULE} <prog module>
<prog module> ::= {<system decls>} <routine list>

A.4 Deklarationen

System-Deklarationen
<system decls> ::= <DEFINE decl> | <TYPE decl> | <var decl>

DEFINE Directive
<DEFINE decl> ::= <DEFINE> <def list>
<def list> ::= <def list>,<def> | <def>
<def> ::= <identifier>=<str const>

TYPE-Deklaration (für Records)
<TYPE decl> ::= TYPE <rec ident list>
<rec ident list> ::= <rec ident list> <rec ident> | <rec ident>
<rec ident> ::= <rec name>=[<field init>]
<rec name> ::= <identifier>
<field init> ::= <fund var decl>

Variablen-Deklarationen
<var decl> ::= <var decl> <base var decl> | <base var decl>
<base var decl> ::= <fund decl> | <POINTER decl> | <ARRAY decl> | <record decl>

Variablen-Deklarationen für elementare Datentypen
<fund decl> ::= <fund decl> <base fund decl> | <base fund decl>
<base fund decl> ::= <fund type> <fund ident list>
<fund type> ::= CARD | CHAR | BYTE | INT
<fund ident list> ::= <fund ident list>,<fund ident> | <fund ident>
<fund ident> ::= <identifier>{=<init opts>}
<init opts> ::= <addr> | [<value>]
<addr> ::= <comp const>
<value> ::= <num const>

Variablen-Deklaration für Pointer
<POINTER decl> ::= <ptr type> POINTER <ptr ident list>
<ptr type> ::= <fund type> | <rec name>
<ptr ident list> ::= <ptr ident list>,<ptr ident> | <ptr ident>
<ptr ident> ::= <identifier>{=<value>}

Variablen-Deklaration für Arrays
<ARRAY decl> ::= <fund type> ARRAY <arr ident list>
<arr ident list> ::= <arr ident list>,<arr ident> | <arr ident>
<arr ident> ::= <identifier>{(<dim>)}{=<arr init opts>}
<dim> ::= <num conat>
<arr init opts> ::= <addr> | [<value>] | <str const>
<addr> ::= <comp const>
<value list> ::= <value list><value> | <value>
<value> ::= <comp const>

Variablen-Deklaration für Records
<record decl> ::= <identifier> <rec ident list>
<rec ident list> ::= <rec ident list>,<rec ident> | <rec ident>
<rec ident> ::= <identifier>{=<address>}
<address> ::= <comp const>

A.5 Variablen-Referenzierung

Speicher-Referenzierung
<mem reference> ::= <mem contents> | @<identifier>
<mem contents> ::= <fund ref> | <arr ref> | <ptr ref> | <rec ref>
<fund ref> ::= <identifier>
<arr ref> ::= <identifier>(<arith exp>)
<ptr ref> ::= <identifier>^
<rec ref> ::= <identifier>.<identifier>

A.6 Action!-Routinen

Routinen
<routine list> ::= <routine list> <routine> | <routine>
<routine> ::= <proc routine> | <func routine>

Strukturen von PROCs
<proc routine> ::= <PROC decl> {<system decls>} {<stmt list>} {RETURN}
<proc decl> ::= PROC <identifier>{=<addr>}({<param decl>})
<addr> ::= <comp const>

Strukturen von FUNCs
<func routine> ::= <FUNC decl> {<system decls>} {<stmt list>} {RETURN (<arith exp>)}
<FUNC decl> ::= <fund type> FUNC <identifier>{.<addr>} ({<param decl>})
<addr> ::= <comp const>

Routinen aufrufen
<routine call> ::= <FUNC call> | <PROC call>
<FUNC call> ::= <identifier>({<params>})
<PROC call> ::= <identifier>({<params>})

Parameter
<param decl> ::= <var decl>

Anmerkung: Maximal sind 8 Parameter zulässig.

A.7 Anweisungen

<stmt list> ::= <stmt list> <stmt> | <stmt>
<stmt> ::= <simp stmt> | <struc stmt> | <code block>
<simp stmt> ::= <assign stmt> | <EXIT stmt> | <routine call>
<struc stmt> ::= <IF stmt> | <DO loop> | <WHILE loop> | <FOR loop>

Zuordnende Anweisungen
<assign stmt> ::= <mem contents>=<arith exp>

EXIT-Anweisung
<EXIT stmt> ::= EXIT

IF-Anweisung
<IF stmt> ::= IF <cond exp> THEN {stmt list} {|:ELSEIF exten:|}{ELSE exten} FI
<ELSEIF exten> ::= ELSEIF <cond exp> THEN {stmt list}
<ELSE exten> ::= ELSE {stmt list}

DO-OD-Schleife
<DO loop> ::= DO {<stmt list>} {<UNTIL stmt>} OD

UNTIL-Anweisung
<UNTIL stmt> ::= UNTIL <cond exp>

WHILE-Schleife
<WHILE loop> ::= WHILE <cond exp> <DO loop>

FOR-Schleife
<FOR loop> ::= FOR <identifier>=<start> TO <finish> {STEP <inc>} <DO loop>
<start> ::= <arith exp>
<finish> ::= <arith exp>
<inc> ::= <arith exp>

Code-Blöcke
<code block> ::= [<comp const list>]
<comp const list> ::= <comp const list> <comp const> | <comp const>

A.8 Ausdrücke

Relationale Ausdrücke
<complex rel> ::= <complex rel><special op><simp rel exp> | <simp
rel exp><special op><simp rel exp>
<simple rel exp> ::= <arith exp><rel op><arith exp>

Arithmetische Ausdrücke
<arith exp> ::= <arith exp><add op><mult exp> | <mult exp>
<mult exp> ::= <mult exp><mult op><value> | <value>
<value> ::= <num const> | <mem reference> | (<arith exp>)

B - Speicherbelegung durch Action!

Adresse	Belegung
$00	OS und Action! Variablen
$CA	Freier Speicher
$CE	Action! Variablen
$D4	ATARI Fließkommaregister
$100	Operating System
$480	Action! Variablen
$580	ATARI Fließkommapuffer
$600	Operating System
MEMLO	Action! Compiler Stacks
MEMLO+$200	Action! Puffer für Editorzeile
MEMLO+$300	Action! Hash-Tabellen
MEMLO+$750	Action! Puffer für Editortext ------------------------------ Action! Compiler Code Speicherbereich
MEMTOP-$800	Action! Compiler Symbol-Tabelle
MEMTOP	Bildschirmspeicher
$A000	Action! Modul
$C000	OS, ROMs, etc.
$FFFF	

Anmerkung: Der Speicherbereich für den Compilercode beginnt dort, wo der Textpuffer des Editors endet. Das ermöglicht eine dynamische Speicherausnutzung für Editor- und Compiler-Puffer. Siehe Kapitel V, Abschnitt 2.

C - Action! Systemadressen

Eine Sammlung von bekannten Systemadressen mit den Erläuterungen dazu.

C.1 Speicherstellen von Compiler und RunTime

Eine Übersicht der wichtigen Speicherstellen, die von Compiler und RunTime verwendet werden. Angegeben werden Adresse (als Hexadezimal- und Dezimalzahl), das interne Label und eine kurze Beschreibung. Sofern das Ändern einer Speicherstelle sinnvoll sein kann, wird das in der Beschreibung angeführt. Und ggf. erfolgt ein Hinweis auf den betreffenden Abschnitt, der mehr Informationen dazu bietet.

Systemadressen, die zwei Bytes lang sind (Low/High-Byte-Format für 6502-Prozessor), können auch Zeiger auf eine Adresse sein.

$0E,$0F (14,15) CODE oder APPMHI[114]

Der von Action! verwendete 'Speicherzähler'. Zeigt auf die Speicherstelle, an die das nächste Byte an Code beim Kompilieren abgelegt wird, und sollte nur geändert werden, bevor ein Programm kompiliert wird[115]. Er kann aber mit Vorsicht auch verändert werden, um Objektcode für den ROM-Bereich zu erzeugen[116]. Nach dem Kompilieren steht hier die Endadresse des kompilierten Programms.

$9B,$9C (155,156) BUF

Adresse des Editorpuffers. Dieser Puffer wird ebenfalls von der PROC OPEN aus der Library des Moduls verwendet, um einen übergebenen Filenamen zu überprüfen und ggf. einen Devicenamen zu ergänzen. Beim Initialisieren muss hier eine gültige Adresse gesetzt werden, wenn das kompilierte Programm zusammen mit dem Modul laufen soll.

Wichtig: Im Gegensatz dazu kann die PROC OPEN der RunTime-Library dies nicht.[117]

$A0-$AF (160-175) ARGS

In diesem Teil der Page 0 werden sowohl Parameter für FUNCs und PROCs als auch Zwischenergebnisse von Ausdrücken abgelegt. Parameter werden aufsteigend ab $A0, Zwischenergebnisse absteigend ab $AF gespeichert.

$B0 (176) STBASE

Höherwertiges Byte der Adresse, an der die Symboltabelle beginnt (Page-Nummer).

$B1,$B2 (177,178) STGLOBAL

Speicherstelle, ab der die 512 Byte große Hash-Tabelle für globale Symbole liegt.

$B3,$B4 (179,180 STLOCAL

Speicherstelle, ab der die 512 Byte große Hash-Tabelle für lokale Symbole liegt.

$B5,$B6 (181,182) CODEOFF

Offset zwischen der Adresse, wo der kompilierte Code gespeichert wird (bestimmt durch CODE; $0E,$0F), und der Adresse, an welcher das fertige Programm laufen soll (CODE+CODEOFF=RUNAD).[118]

---

114) Das ATARI Profibuch – ABBUC-Edtition, S. 5.
115) Kapitel VII, Abschnitt 5.1.
116) Kapitel VII, Abschnitt 6.1.
117) Kapitel VII, Abschnitt 6.2.4.
118) Kapitel VII, Abschnitt 5.2.

$B7 (183) DEVICE
Kanalnummer des derzeit voreingestellten Geräts. Wird der Wert geändert auf eine zuvor per OPEN geöffnete Kanalnummer, wird die Ausgabe durch die Library-Routine auf diesen Kanal umgeleitet.[119]

$CE,$CF (206,207) STG2
Wird nur verwendet, wenn das Flag in $04C4 für die große Symboltabelle gesetzt ist. Hier ist die Adresse der 512 Byte großen Hash-Tabelle gespeichert, welche die zweite Hälfte der Namen für die globalen Symbole enthält.

$0491,$0492 (1169,1170) CODEBASE
Erste Adresse, ab der Code beim Kompilieren gespeichert wird. Sie wird für den späteren Gebrauch mit dem W-Kommando bewahrt.

$0493,$0494 (1171,1172) CODESIZE
Enthält die Anzahl an Bytes, die beim Kompilieren erzeugt wurden.

$0495 (1173) STSP
Enthält die Anzahl an 256-Byte-Pages, die für die lokalen und globalen Symboltabellen reserviert sind. Standardeinstellung ist 8 Pages = 2KiB.

$049A (1178) LIST
Flag, mit dem sich die List-Option[120] für den Compiler schalten lässt. Kann auch beim Kompilieren jederzeit per SET geschaltet werden.

$04AD (1197) FRSTCHAR
Das hier gespeicherte Zeichen bestimmt die Stelle, an der die Symboltabelle in zwei Hälften geteilt wird, wenn das Flag BIGST gesetzt ist. Siehe auch $04C4, BIGST.

$04C0 (1216) BCKGRND
Hintergrundfarbe setzen (auf eigenes Risiko).

$04C4 (1220) BIGST
Wenn dieses Flag gesetzt ist, wurde die Symboltabelle für globale Variablen in zwei Hälften aufgeteilt. Dann stehen statt 255 bis zu 510 globale Symbole zur Verfügung.[121]

$04CB (1227) ERROR
Ein JMP zur aktuellen Routine für die Fehlerbehandlung. Für den Compiler ist dies die Adresse der PROC Error. Kann durch eine eigene Routine ersetzt werden.[122]

$0500-$05FF (1280-1535)
Ein Puffer, der von der OPEN-Routine der RunTime verwendet wird. Wird OPEN ein Filenamen übergeben, wird er hierher verschoben und bekommt ein RETURN ($9B) angehängt. Der Speicherbereich für diesen Puffer kann selbstfestgelegt werden, indem die DEFINEs am Anfang der RunTime entsprechend geändert werden.

---
119) Kapitel VI, Abschnitt 4.
120) Kapitel III, Abschnitt 2.5.
121) Kapitel VII, Abschnitt 4.2.
122) Kapitel VI, Abschnitt 7.3 und Kapitel VII, Abschnitt 6.2.2.

Die Library-Routine im Modul verwendet stattdessen einen Puffer, auf den durch BUF ($9B) verwiesen wird.

C.2 Weitere Adressen von Action!

Page 0
Action! nutzt die Adressen $80-$CA und $CE-$D4[123].

$A0,$A1 160,161
Während des Programmablaufs werden hier die von FUNCs zurückgelieferten Werte abgelegt.

$A3-$AF 163-175
Hier speichert der Compiler die Byte-Werte 4 – 16 für übergebene Parameter. Die Byte-Werte 1-3 werden in den CPU-Registern A, X und Y übergeben ($A0-$A2; 160-162).

Page 4

$04CE (1230) W1SIZE
Größe von Fenster 1 (s. Optionen).

$04C3 (1219) TRACE
Interne Variable "TRACE".

Page 5
$05C0 EOF
Interne Variable EOF (8 Bytes).

123) Das ATARI Profibuch – ABBUC-Edtition, S. 17.

Raum für eigene Notizen

D - Fehlercodes

Code	Erläuterungen
0	Kein Speicher mehr frei. Siehe II.4.3 und V.4.4.
1	Im String fehlt ein ".
2	Verschachtelte DEFINEs. Nicht zulässig.
3	Tabelle für globale Variablen ist voll.
4	Tabelle für lokale Variablen ist voll.
5	Syntaxfehler in der SET-Direktive.
6	Fehler in der Deklaration. Z.B. falsches Format.
7	Zu viele Argumente.
8	Nicht deklarierte Variable.
9	Keine Konstante.
10	Nicht erlaubte Zuweisung.
11	Unbekannter Fehler.
12	THEN fehlt.
13	FI fehlt.
14	Kein Codespeicher mehr. Siehe V.4.4!
15	DO fehlt.
16	TO fehlt.
17	Ausdruck oder Format falsch.
18	Offene Klammer.
19	OD fehlt.
20	Kann Speicher nicht mehr zuweisen.
21	Ungültige ARRAY-Bezeichnung.
22	Datei ist für Input zu lang. In kleinere Teile aufspalten.
23	Ungültiger bedingter Ausdruck
24	Ungültige FOR-Anweisung-Konstruktion
25	Ungültiges EXIT. Keine DO-OD-Schleife!
26	Mehr als 16 Schachtelungen.
27	Ungültige TYPE-Konstruktion.
28	Ungültiges RETURN.
61	Kein Speicher mehr für die Symboltabelle.[124]
128	<BREAK>-Taste wurde zum Programmabbruch benutzt.

Hinweis:
Trat beim Kompilieren eines Programms ein Fehler auf, steht der Cursor beim nächsten Aufruf des Editors an der Stelle im Programmtext, an welcher der Fehler entdeckt wurde, oft aber auch in der Zeile nach der Fehlerstelle. Also sorgfältig prüfen.

124) Abhilfe: Kapitel V, Abschnitt 2.6 und Kapitel VII, Abschnitt 4.

E - Übersicht der Editor-Kommandos

Kommandos wie beim ATARI-Editor

<SHIFT>	Umschaltung für Großbuchstaben, andere Zeichen oder Kommandos
<CTRL>	erzeugen von Kommandos oder Spezialzeichen
<ATARI/INVERS>	inverse Darstellung ein-/ausschalten
<ESC>	für Eingabe von Steuerzeichen
<CAPS>	Umschaltung auf Kleinbuchstaben
<SHIFT><CAPS>	Umschaltung auf Großbuchstaben
<CTRL><INSERT>	Leerzeichen einfügen
<CTRL><DELETE>	Zeichen löschen
<SHIFT><INSERT>	Leerzeile einfügen
<SHIFT><SET TAB>	Tabulatorstopp setzen
<CTRL><CLR TAB>	Tabulatorstopp löschen
<CTRL><↑> oder <F1>	Cursor hoch
<CTRL><↓> oder <F2>	Cursor 'runter
<TAB>	zum nächsten Tabulator springen

Abweichende Funktion von Kommandos wie beim ATARI-Editor

<BREAK>	nicht belegt
<SHIFT><CLEAR>	gesamten Text im aktiven Fenster löschen
<RETURN>	Modus Überschreiben: nächste Zeile Modus Einfügen: RETURN-Zeichen einfügen
<SHIFT><DELETE>	Zeile löschen und in Kopierpuffer übertragen
<BK SP>	Modus Überschreiben: Zeichen links vom Cursor löschen Modus Einfügen: zusätzlich Rest der Zeile heranrücken
<CTRL><←> oder <F3>	Cursor links
<CTRL><→> oder <F4>	Cursor rechts erreicht der Cursor bei diesen beiden Kommandos den Rand des Fensters, wird die Zeile 'durchgeschoben', falls Anfang/Ende außerhalb des Fensters liegen

Zusätzliche Kommandos des Action!-Editors

I/O-Kommandos

<CTRL><SHIFT> <R> filespec	Datei lesen
<CTRL><SHIFT> R ?n:*.*	Directory of Dn:
<CTRL><SHIFT> W, filespec	Datei schreiben
<CTRL><SHIFT> W, P:	ausdrucken

Cursorbewegung im Fenster

<CTRL><SHIFT> <	Cursor an Zeilenanfang
<CTRL><SHIFT> >	Cursor an Zeilenende
<CTRL><SHIFT> E	Dateiende
<CTRL><SHIFT> H	Dateianfang
<CTRL><SHIFT> <↑>	ein Bild nach oben
<CTRL><SHIFT> <↓>	ein Bild nach unten

<CTRL><SHIFT> <[>	Bild nach links scrollen
<CTRL><SHIFT> <]>	Bild nach rechts scrollen
<CTRL><SHIFT> <G>	zur Markierung springen

Markierungen

<CTRL><SHIFT> <T>	Markierung setzen
<CTRL><SHIFT> <G>	zur Markierung springen

Texteingabe

Programm eingeben	einfach Text schreiben

Text Einfügen/Überschreiben

<CTRL><SHIFT> I	Modus umschalten

Vorherige Zeile zurückholen

<CTRL><SHIFT> U	Cursor nicht bewegen und alte Zeile wiederherstellen
<CTRL><SHIFT> P	Cursor nicht bewegen und Zeile zurückrufen

Blockoperationen

<SHIFT><Delete>	zeilenweises ausschneiden und in Puffer übernehmen
<CTRL><SHIFT> P	Pufferinhalt einfügen

Suchen/Ersetzen

<CTRL><SHIFT> F	zu findenden String eingeben
<CTRL><SHIFT> S	ersetzen durch Eingabe neuer String,
<RETURN>	alter String

Trennen und Zusammenfügen von Zeilen

<CTRL><SHIFT> <RETURN>	Cursor positionieren und trennen
<CTRL><SHIFT> <Bk Sp>	Cursor an Anfang der 2. Zeile und zusammenfügen

Editorfenster

<CTRL><SHIFT><2>	Editorfenster 2 öffnen bzw. von Fenster 1 zu Fenster 2 wechseln
<CTRL><SHIFT><1>	von Fenster 2 zu Fenster 1 wechseln
<CTRL><SHIFT><D>	aktives Fenster löschen; Fenster 2 wird dann zu 1.

Editor verlassen

<CTRL><SHIFT> M	Editor verlassen (führt in den Monitor)

F - Übersicht der Monitor-Kommandos

B	Boot → Neustart
C {"<filespec>"}	Compile →kompilieren
D	DOS
E	Editor
O	Options-Menü
P	Proceed →Fortführen nach Stop
R {"<filespec">}	Run →Programm starten
SET <address> = <value>	wie Poke in BASIC
W {"<filespec">}	Write →speichern
X <statement>I:, <statement>:I	Statement(s) ausführen
? <address>	wie Peek auch für Compiler-Konstanten
* <address>	'peekt' ab Adresse alle Speicherstellen oder Compiler-Konstanten

G - Übersicht zum Options-Menü

Aufforderung	Status	Option	Erläuterung
Display on?	Y (an)	Y oder N	Steuert den Bildschirm beim Kompilieren und bei I/O-Operationen
Bell off?	N (aus)	Y oder N	Warnton an/aus
Case insensitive?	N (aus)	Y oder N	Compilerprüfung auf Unterschied von Groß-/Kleinschreibung an/aus
Trace on?	N (aus)	Y oder N	Trace-Funktion an/aus
List on?	N (aus)	Y oder N	Listet das Programm während des Kompilierens auf dem Bildschirm
Window size?	18	5 - 18	Steuert die Größe von Fenster 1. Fenster 1 und 2 belegen zusammen 23 Zeilen.
Line size	120	1 - 240	Steuert die Zeilenlänge
Left margin?	2	0 - 39	Setzt den linken Rand
EOL character?	$9B	jedes ATASCII-Zeichen	Zeichen für Zeilenende einstellen

H - Benchmark Primzahlen

```
; BENCHMARKTEST FUER Action!
; Ausgabe der Primzahlen. (C) 1983 by ACS

  DEFINE size = "8190",
         ON = "1",
         OFF = "0"

  BYTE ARRAY flags(size+1)

  CARD count, i, k, prime

  BYTE DISPLAY=$22F,
       iter,
       tick=20,
       tock=19

PROC Primes()
  DISPLAY = 0 ;comment this line to leave display on
  tick = 0
  tock = 0
  FOR iter=1 TO 10
     DO
     count = 0
     ; turn flags on (non-zero)
     SetBlock(flags, size, ON)
     FOR i = 0 TO size
        DO
        IF flags(i) THEN
           prime = i+i+3
           ;PrintCE(prime)  ;Uncomment to print primes
           k = prime + i
           WHILE k <= size
              DO
              flags(k) = OFF
              k ==+ prime
              OD
           count ==+ 1
        FI
        OD
     OD
  i=tick+256+tock
  DISPLAY = $22 ;turn display back on
  PrintF("%U Primes done in %U ticks %E", count, i)
RETURN
```

I - Vergleich ATARI-BASIC – Action!

Die Action!-Beispiele setzen folgende Variablendeklaration voraus:

```
INT i,j,k      BYTE ARRAY s,t,aa,ba
CARD c,d,e     CARD ARRAY ca,da,ea
BYTE a,b       INT ARRAY ia,ja,ka
```

```
BASIC                         Action!
--------------------------    -----------------------
C=D+I*A                       c = d + i *a

IF A<>0 THEN B=1              IF a<>0 THEN b=1 FI

10 IF A=0 THEN 30             IF a<>0 THEN
20 B=1 : C=A*2                  b=1  c=a*2
30 REM                        FI

10 IF A=0 THEN B=1 GOTO 30      IF a=0 THEN b=1
20 B=7                        ELSE b=7
30 REM                        FI

FOR I=1 TO 100...             FOR i = 1 TO 100 DO ...
NEXT I                        OD

PRINT "HELLO"                 PrintE("HELLO")

PRINT "HELLO";                Print("HELLO")

PRINT #5;"HELLO"              PrintDE(5,"HELLO")

PRINT #5;"HELLO";             PrintD(5,"HELLO")

PRINT I                       PrintIE(i)

PRINT "I=";I                  PrintF("I=%I%E", i)
                                    oder
                              Print("I=")  PrintIE(i)

PRINT #3; B*3;                PrintBD(3, b*3)

INPUT I                       Put('?) : i=Input()
```

Anmerkung: Doppelpunkte werden von Action! ignoriert und sind daher als Trenner zwischen Anweisungen verwendbar.

```
INPUT B$                      Put('?) : InputS(ba)
```

```
PUT #0,65                  Put('A)  oder
                           Put(65)  oder
                           Put($41)
GET #C,B                   b= GetD(c)

OPEN #1,4,0,"K:"           Open(1, "K:", 4, 0)

CLOSE #3                   Close(3)

NOTE #1,C,B                Note (1, @c, @b)

POINT #1,C,B               Point(1, c, b)

XIO 18,#6,0,0,"S:"         XIO(6,0,18,0,0,"S:")
oder die Library-Routine Fill benutzen

B=PEEK(C)                  b = Peek(c)
oder besseres Action!
                           ba = c : b = ba^

POKE C,B                   Poke(c,b)
oder besseres Action!
                           ba = c : ba^ = b

GRAPHICS 8                 Graphics(8)

COLOR 3                    color = 3
```

Anmerkung: color ist eine Variable der System-Library und in Action! vordefiniert

```
DRAWTO C,D                 DrawTo(c,d)

LOCATE C,D,B               b = Locate(c,d)

PLOT C,D                   Plot(c,d)

POSITION C,D               Position(c,d)
SETCOLR 0,1,C              SetColor(0,1,c)

GRAPHICS 24 : COLOR C :    Graphics(24) : color = c
PLOT 200,150 :             Plot(200,150)
DRAWTO 120,20 :            DrawTo(120,20)
POSITION 40,150 :          Fill(40,150)
POKE 765,C :
XIO 18,#6,0,0,"S:"

SOUND 0,121,10,6           Sound(0,121,10,6)
```

```
C = PADDLE(B)             c = Paddle(b)
C = PTRIG(B)              c = Ptrig(b)
C = STICK(B)              c = Stick(b)
C = STRIG(B)              c = Strig(b)
B$ = S$                   SCopy(ba, s)

B$ = S$(3,5)              SCopyS(ba, s, 3 ,5)

B$(3,5) = S$              SAssign(ba, s, 3, 5)

B=INT(6*RND(0)) + 1       b = Rand(6) + 1

FOR C = 4000 TO 5000 :
POKE C,0 : NEXT C         Zero(4000,1001)

STOP                      Break()

B$ = STR$(I)              StrI(i, ba)

I = VAL(S$)               i = ValI(s)
```

J - Runtime Library

Der nachfolgende Programmtext der Runtime Library von Optimized Systems Software wurde durch Erhard aufgearbeitet und kommentiert. Der Prozess ist noch nicht abgeschlossen, trotzdem hier der aktuelle Stand. Die Gliederung zeigt die erste Minitabelle. Die innerhalb der Runtime Library bestehenden Abhängigkeiten, markiert durch punktierte Unterstreichung, gilt es zu beachteten, wenn die Runtime Library für eigene Zwecke angepasst werden soll.

Adresse	**Routine**
Source	**Source**

$6000	MODULE ; SYS.ACT ; (c) 1983,1984 ACS ; Copyright (c) 1983,1984 ; by Action Computer Services (ACS) ; All rights reserved. ; ; version 1.4 ; last modified March 27, 1984 DEFINE STRING="CHAR ARRAY" DEFINE EOL="$9B" DEFINE OpenBuf = "$0500" DEFINE OpenBufL = "$00" DEFINE OpenBufH = "$05" STRING copy_right(0) = "(c)1983 Action Computer Services"
*= $6000 ; .BYTE $01,$20	

$6002	;Primitive IO routines PROC Clos=*(BYTE d) [$FFA2$A686$CA0$AD0]

P_CLOS LDX #$FF ; A2 FF
        STX L00A6 ; 86 A6
        LDY #$0C ; A0 0C
        BNE L6014 ; D0 0A	

$600A	PROC Output=*(BYTE d,STRING s) [$A684$BA0$4D0]

P_OUTPUT STY L00A6 ; 84 A6
        LDY #$0B ; A0 0B
        BNE L6014 ; D0 04	

$6010	PROC In=*(BYTE d,STRING s) [$A684$5A0$A586$A2$0$A386]

P_IN STY L00A6 ; 84 A6
        LDY #$05 ; A0 05
L6014 STX L00A5 ; 86 A5
        LDX #$00 ; A2 00
        STX L00A3 ; 86 A3	

```
$601A   PROC XIOstr=*(BYTE d,x,c,a1,a2,STRING s)
        [$A0A$A0A$98AA$9D$342$A3A5$AF0$9D$34A$A4A5$9D$34B$A9$0$9DA8$349$A5B1
        $9D$348$12F0$18$A5A5$169$9D$344$A6A5$69$0$9D$345$4C$E456$60]

P_XIOSTR    ASL A           ; 0A
            ASL A           ; 0A
            ASL A           ; 0A
            ASL A           ; 0A
            TAX             ; AA
            TYA             ; 98
            STA ICCOM,X     ; 9D 42 03
            LDA L00A3       ; A5 A3
            BEQ L6031       ; F0 0A
            STA ICAX1,X     ; 9D 4A 03
            LDA L00A4       ; A5 A4
            STA ICAX2,X     ; 9D 4B 03
            LDA #$00        ; A9 00
L6031       TAY             ; A8
            STA ICBLH,X     ; 9D 49 03
            LDA (L00A5),Y   ; B1 A5
            STA ICBLL,X     ; 9D 48 03
            BEQ L604E       ; F0 12
            CLC             ; 18
            LDA L00A5       ; A5 A5
            ADC #$01        ; 69 01
            STA ICBAL,X     ; 9D 44 03
            LDA L00A6       ; A5 A6
            ADC #$00        ; 69 00
            STA ICBAH,X     ; 9D 45 03
            JMP CIOV        ; 4C 56 E4
L604E       RTS             ; 60
```

```
$604F   PROC Opn=*(BYTE d,STRING s,BYTE m,o)
        [$A586$A684$3A0$4CXIOstr]

P_OPN       STX L00A5       ; 86 A5
            STY L00A6       ; 84 A6
            LDY #$03        ; A0 03
            JMP P_XIOSTR    ; 4C 1A 60
```

```
$6058   PROC Prt=*(BYTE d,STRING s)
        [$A586$A684$A2$0$A386$9A0$20XIOstr$AD0$BA9$9D$342$9BA9$4C$E456$60]

P_PRT       STX L00A5       ; 86 A5
            STY L00A6       ; 84 A6
            LDX #$00        ; A2 00
            STX L00A3       ; 86 A3
            LDY #$09        ; A0 09
            JSR P_XIOSTR    ; 20 1A 60
            BNE L6071       ; D0 0A
            LDA #$0B        ; A9 0B
            STA ICCOM,X     ; 9D 42 03
            LDA #$9B        ; A9 9B
            JMP CIOV        ; 4C 56 E4
L6071       RTS             ; 60
```

```
        PROC Error(BYTE err)
        [$6C$A$0$1113$8301]

L6072       CLC             ; 18
P_ERROR()   JMP L6076       ; 4C 76 60
L6076       STA L6072       ; 8D 72 60
            JMP (DOSVEC)    ; 6C 0A 00
            .BYTE $13,$11,$01,$83
```

```
$6080   PROC Break=*()
        [$BA$8E$4C1$80A0$98$4C Error]

P_BREAK     TSX             ; BA
            STX L04C1       ; 8E C1 04
            LDY #$80        ; A0 80
            TYA             ; 98
            JMP P_ERROR()   ; 4C 73 60
```

$608A	;math library routines PROC LShift=*() [$84A4$AF0$8586$A$8526$88$FAD0$85A6$60]				
P_LSHIFT	LDY L0084	; A4 84		DEY	; 88
	BEQ L6098	; F0 0A		BNE L6090	; D0 FA
	STX L0085	; 86 85		LDX L0085	; A6 85
L6090	ASL A	; 0A	L6098	RTS	; 60
	ROL L0085	; 26 85			

$6099	PROC RShift=*() [$84A4$AF0$8586$8546$6A$88$FAD0$85A6$60]				
P_RSHIFT	LDY L0084	; A4 84		DEY	; 88
	BEQ L60A7	; F0 0A		BNE L609F	; D0 FA
	STX L0085	; 86 85		LDX L0085	; A6 85
L609F	LSR L0085	; 46 85	L60A7	RTS	; 60
	ROR A	; 6A			

$60A8 $60AC	PROC SetSign=*() [$D3A4$1010] PROC SS1=*() [$8685$8786$38$A9$0$86E5$A8$A9$0$87E5AA98$60]				
P_SETSIGN	LDY L00D3	; A4 D3		TAY	; A8
	BPL L60BC	; 10 10		LDA #$00	; A9 00
P_SS1	STA L0086	; 85 86		SBC L0087	; E5 87
	STX L0087	; 86 87		TAX	; AA
	SEC	; 38		TYA	; 98
	LDA #$00	; A9 00	L60BC	RTS	; 60
	SBC L0086	; E5 86			

$60BD	PROC SMOps=*() [$D386$E0$0$310$20SS1$8285$8386$85A5$E10$AA$D345$D385$84A5$20SS1$8485$8586$A9$0$8785$60]				
P_SMOPS	STX L00D3	; 86 D3		EOR L00D3	; 45 D3
	CPX #$00	; E0 00		STA L00D3	; 85 D3
	BPL L60C6	; 10 03		LDA L0084	; A5 84
	JSR P_SS1	; 20 AC 60		JSR P_SS1	; 20 AC 60
L60C6	STA L0082	; 85 82		STA L0084	; 85 84
	STX L0083	; 86 83		STX L0085	; 86 85
	LDA L0085	; A5 85	L60DC	LDA #$00	; A9 00
	BPL L60DC	; 10 0E		STA L0087	; 85 87
	TAX	; AA		RTS	; 60

$60E1	PROC MultB=*() [$1BF0$CA$C786$AA$15F0$C686$A9$0$8A2$A$C606$290$C765$CA$F6D0$18$8765$8785$86A5$87A6$60]				
P_MULTB	BEQ L60FE	; F0 1B		BCC L60F6	; 90 02
	DEX	; CA		ADC L00C7	; 65 C7
	STX L00C7	; 86 C7	L60F6	DEX	; CA
	TAX	; AA		BNE L60EF	; D0 F6
	BEQ L60FE	; F0 15		CLC	; 18
	STX L00C6	; 86 C6		ADC L0087	; 65 87
	LDA #$00	; A9 00		STA L0087	; 85 87
	LDX #$08	; A2 08	L60FE	LDA L0086	; A5 86
L60EF	ASL A	; 0A		LDX L0087	; A6 87
	ASL L00C6	; 06 C6		RTS	; 60

$6103	PROC MultI=*() [$20SMOps$82A6$1BF0$C686$84A6$15F0CAC786$8A2$A8726C606$690$C765$290$87E6CAF0D0$8685$82A5$85A6$20MultB$83A5$84A6$20MultB$4CSetSign]

```
P_MULTI     JSR P_SMOPS    ; 20 BD 60
            LDX L0082      ; A6 82
            BEQ L6125      ; F0 1B
            STX L00C6      ; 86 C6
            LDX L0084      ; A6 84
            BEQ L6125      ; F0 15
            DEX            ; CA
            STX L00C7      ; 86 C7
            LDX #$08       ; A2 08
L6115       ASL A          ; 0A
            ROL L0087      ; 26 87
            ASL L00C6      ; 06 C6
            BCC L6122      ; 90 06
            ADC L00C7      ; 65 C7
            BCC L6122      ; 90 02
            INC L0087      ; E6 87
L6122       DEX            ; CA
            BNE L6115      ; D0 F0
L6125       STA L0086      ; 85 86
            LDA L0082      ; A5 82
            LDX L0085      ; A6 85
            JSR P_MULTB    ; 20 E1 60
            LDA L0083      ; A5 83
            LDX L0084      ; A6 84
            JSR P_MULTB    ; 20 E1 60
            JMP P_SETSIGN  ; 4C A8 60
```

$6138	PROC DivI=*() [$20SMOps$85A5$27F0$8A2$8226$8326$8726$38$83A5$84E5$A8$87A5$85E5$490$8785$8384CAE7D0$82A5$2A$A2$0$83A4$8684$4CSetSign$10A2$8226$8326$2A$4B0$84C5$390$84E5$38CAEFD0$8226$8326$8685$82A5$83A6$4CSetSign]

```
P_DIVI      JSR P_SMOPS    ; 20 BD 60
            LDA L0085      ; A5 85
            BEQ L6166      ; F0 27
            LDX #$08       ; A2 08
L6141       ROL L0082      ; 26 82
            ROL L0083      ; 26 83
            ROL L0087      ; 26 87
            SEC            ; 38
            LDA L0083      ; A5 83
            SBC L0084      ; E5 84
            TAY            ; A8
            LDA L0087      ; A5 87
            SBC L0085      ; E5 85
            BCC L6157      ; 90 04
            STA L0087      ; 85 87
            STY L0083      ; 84 83
L6157       DEX            ; CA
            BNE L6141      ; D0 E7
            LDA L0082      ; A5 82
            ROL A          ; 2A
            LDX #$00       ; A2 00
            LDY L0083      ; A4 83
            STY L0086      ; 84 86
            JMP P_SETSIGN  ; 4C A8 60
L6166       LDX #$10       ; A2 10
L6168       ROL L0082      ; 26 82
            ROL L0083      ; 26 83
            ROL A          ; 2A
            BCS L6173      ; B0 04
            CMP L0084      ; C5 84
            BCC L6176      ; 90 03
L6173       SBC L0084      ; E5 84
            SEC            ; 38
L6176       DEX            ; CA
            BNE L6168      ; D0 EF
            ROL L0082      ; 26 82
            ROL L0083      ; 26 83
            STA L0086      ; 85 86
            LDA L0082      ; A5 82
            LDX L0083      ; A6 83
            JMP P_SETSIGN  ; 4C A8 60
```

$6186	PROC RemI=*() [$20 DivI$86A5$87A6$60]

```
P_REMI      JSR P_DIVI     ; 20 38 61
            LDA L0086      ; A5 86
            LDX L0087      ; A6 87
            RTS            ; 60
```

```
$618E    PROC SArgs=*()
         [$A085$A186$A284$18$68$8485$369$A8$68$8585$69$0$48$98$48$1A0$84B1$8
         285$C8$84B1$8385$C8$84B1$A8$B9$A0$0$8291$88$F810$11A5$FD0$11E6$4C
         Break$6308$1109$1819$2113$3323$60]
```

```
P_SARGS     STA FRET         ; 85 A0
            STX FRET+1       ; 86 A1
            STY BPTR2        ; 84 A2
            CLC              ; 18
            PLA              ; 68
            STA L0084        ; 85 84
            ADC #$03         ; 69 03
            TAY              ; A8
            PLA              ; 68
            STA L0085        ; 85 85
            ADC #$00         ; 69 00
            PHA              ; 48
            TYA              ; 98
            PHA              ; 48
            LDY #$01         ; A0 01
            LDA (L0084),Y    ; B1 84
            STA L0082        ; 85 82
            INY              ; C8
            LDA (L0084),Y    ; B1 84
            STA L0083        ; 85 83
            INY              ; C8
            LDA (L0084),Y    ; B1 84
            TAY              ; A8
L61B2       LDA FRET,Y       ; B9 A0 00
            STA (L0082),Y    ; 91 82
            DEY              ; 88
            BPL L61B2        ; 10 F8
            LDA BRKKEY       ; A5 11
            BNE L61CD        ; D0 0F
            INC BRKKEY       ; E6 11
            JMP P_BREAK      ; 4C 80 60
            .WORD $6308
            .WORD $1109
            .WORD $1819
            .WORD $2113
            .WORD $3323
L61CD       RTS              ; 60
```

```
$61CE    SET $4E4=LShift
         SET $4E6=RShift
         SET $4E8=MultI
         SET $4EA=DivI
         SET $4EC=RemI
         SET $4EE=SArgs

         PROC ChkErr=*(BYTE r,b,eC)
         [$1610$88C0$8F0$98$80C0$12F0$4C Error$8A$4A4A$4A4A$98AA$9D EOF$60]
```

```
P_CHKERR    BPL L61E6        ; 10 16
            CPY #$88         ; C0 88
            BEQ L61DC        ; F0 08
            TYA              ; 98
            CPY #$80         ; C0 80
            BEQ L61EB        ; F0 12
            JMP P_ERROR()    ; 4C 73 60
L61DC       TXA              ; 8A
            LSR A            ; 4A
            LSR A            ; 4A
            LSR A            ; 4A
            LSR A            ; 4A
            TAX              ; AA
            TYA              ; 98
            STA EOF,X        ; 9D C0 05
L61E6       RTS              ; 60
```

```
$61E7    PROC Break1=*(BYTE err)
         [$1A2$1186$48$20 Break$68$A8$60]
```

```
P_BREAK1    LDX #$01         ; A2 01
            STX BRKKEY       ; 86 11
L61EB       PHA              ; 48
            JSR P_BREAK      ; 20 80 60
            PLA              ; 68
            TAY              ; A8
            RTS              ; 60
```

$61F2	PROC Open=*(BYTE d,STRING f,BYTE m,a2) [48A186$A284$A8$A9$0$99 EOF$A8$A1B1$8D OpenBuf $A8$C8$9BA9$2D0$A1B1$99 OpenBuf 88F8D0$68$A2 OpenBufL $A0 OpenBufH $20Opn$4C ChkErr]

```
P_OPEN      PHA               ; 48
            STX FRET+1        ; 86 A1
            STY BPTR2         ; 84 A2
            TAY               ; A8
            LDA #$00          ; A9 00
            STA EOF,Y         ; 99 C0 05
            TAY               ; A8
            LDA (FRET+1),     ; B1 A1
            STA OPENBUF       ; 8D 00 05
            TAY               ; A8
            INY               ; C8
            LDA #$9B          ; A9 9B
            BNE L620B         ; D0 02
L6209       LDA (FRET+1),Y    ; B1 A1
L620B       STA OPENBUF,Y     ; 99 00 05
            DEY               ; 88
            BNE L6209         ; D0 F8
            PLA               ; 68
            LDX # <OPENBUF    ; A2 00
            LDY # >OPENBUF    ; A0 05
            JSR P_OPN         ; 20 4F 60
            JMP P_CHKERR      ; 4C CE 61
```

$621C	PROC PrintE=*(STRING s) [$A186$AA$A1A4$A5device] PROC PrintDE=*(BYTE d,STRING s) [$20 Prt$4C ChkErr]

```
P_PRINTE    STX FRET+1        ; 86 A1
            TAX               ; AA
            LDY FRET+1        ; A4 A1
P_PRINTDE   LDA DEVICE        ; A5 B7
            JSR P_PRT         ; 20 58 60
            JMP P_CHKERR      ; 4C CE 61
```

$6229	PROC Close=*(BYTE d) [$20 Clos$4C ChkErr]

```
P_CLOSE     JSR P_CLOS        ; 20 02 60
            JMP P_CHKERR      ; 4C CE 61
```

$622F $6236	PROC Print=*(STRING s) [$A186$AA$A1A4$A5device] PROC PrintD=*(BYTE d,STRING s) [$20Output$4C ChkErr]

```
P_PRINT     STX FRET+1        ; 86 A1
            TAX               ; AA
            LDY FRET+1        ; A4 A1
            LDA DEVICE        ; A5 B7
P_PRINTD    JSR P_OUTPUTQ     ; 20 0A 60
            JMP P_CHKERR      ; 4C CE 61
```

$623C	PROC InS=*() [$20In$A084BD348$3F0$38$1E9$A0$0$A591$A0A4$60]

```
P_INS       JSR P_IN          ; 20 10 60
            STY FRET          ; 84 A0
            LDA ICBLL,X       ; BD 48 03
            BEQ L6249         ; F0 03
            SEC               ; 38
            SBC #$01          ; E9 01
L6249       LDY #$00          ; A0 00
            STA (L00A5),Y     ; 91 A5
            LDY FRET          ; A4 A0
            RTS               ; 60
```

```
$6250    PROC InputS=*(STRING s)
         [$A286$AA$A2A4$A5device]
$6257    PROC InputSD=*(BYTE d,STRING s)
         [$48$FFA9$A385$68]
$625D    PROC InputMD=*(BYTE d,STRING s,BYTE m)
         [$48$A186$A284$A0$0$A3A5$A191$68$A2A4]
$626B    PROC InputD=*(BYTE d,STRING s)
         [$20InS$4C ChkErr]
```

```
P_INPUTS    STX BPTR2        ; 86 A2
            TAX              ; AA
            LDY BPTR2        ; A4 A2
            LDA DEVICE       ; A5 B7
P_INPUTSD   PHA              ; 48
            LDA #$FF         ; A9 FF
            STA L00A3        ; 85 A3
            PLA              ; 68
P_INPUTMD   PHA              ; 48
            STX FRET+1       ; 86 A1
            STY BPTR2        ; 84 A2
            LDY #$00         ; A0 00
            LDA L00A3        ; A5 A3
            STA (FRET+1),Y   ; 91 A1
            PLA              ; 68
            LDY BPTR2        ; A4 A2
P_INPUTD    JSR P_INS        ; 20 3C 62
            JMP P_CHKERR     ; 4C CE 61
```

```
$6271    CHAR FUNC GetD=*(BYTE d)
         [$7A2]
$6273    PROC CCIO=*()
         [$A386$A0A$A0A$AA$A3A5$9D$342$A9$0$9D$348$9D$349$98$20$E456$A085$4C
         ChkErr]
```

```
F_GETD      LDX #$07         ; A2 07
P_CCIO      STX L00A3        ; 86 A3
            ASL A            ; 0A
            ASL A            ; 0A
            ASL A            ; 0A
            ASL A            ; 0A
            TAX              ; AA
            LDA L00A3        ; A5 A3
            STA ICCOM,X      ; 9D 42 03
            LDA #$00         ; A9 00
            STA ICBLL,X      ; 9D 48 03
            STA ICBLH,X      ; 9D 49 03
            TYA              ; 98
            JSR CIOV         ; 20 56 E4
            STA FRET         ; 85 A0
            JMP P_CHKERR     ; 4C CE 61
```

```
$6290    PROC PutE=*()
         [$A9$9B]
$6292    PROC Put=*(CHAR c)
         [$AA$A5device]
$6295    PROC PutD=*(BYTE d,CHAR c)
         [$A186$A1A4]
$6299    PROC PutD1=*()
         [$BA2$4C CCIO]
```

```
P_PUTE      LDA #$9B         ; A9 9B
P_PUT       TAX              ; AA
            LDA DEVICE       ; A5 B7
P_PUTD      STX FRET+1       ; 86 A1
            LDY FRET+1       ; A4 A1
P_PUTD1     LDX #$0B         ; A2 0B
            JMP P_CCIO       ; 4C 73 62
```

```
$629E    PROC PutDE=*(BYTE dev)
         [$A0$9B$F7D0]
```

```
P_PUTDE     LDY #$9B         ; A0 9B
            BNE P_PUTD1      ; D0 F7
```

```
$62A2    PROC XIO=*(BYTE d,f,c,a1,a2,STRING s)
         [$20XIOstr$4C ChkErr]
```

```
P_XIO       JSR P_XIOSTR     ; 20 1A 60
            JMP P_CHKERR     ; 4C CE 61
```

$62A8	PROC CToStr=*() [$D485$D586$20$D9AA20D8E6$FFA0$A2$0$C8$E8$F3B1$9D$550$F710$8049$9D$550$8E$550$60]

```
P_CTOSTR    STA L00D4       ; 85 D4
            STX L00D5       ; 86 D5
            JSR IFP         ; 20 AA D9
            JSR FASC        ; 20 E6 D8
            LDY #$FF        ; A0 FF
            LDX #$00        ; A2 00
L62B6       INY             ; C8
            INX             ; E8
            LDA (L00F3),Y   ; B1 F3
            STA L0550,X     ; 9D 50 05
            BPL L62B6       ; 10 F7
            EOR #$80        ; 49 80
            STA L0550,X     ; 9D 50 05
            STX L0550       ; 8E 50 05
            RTS             ; 60
```

$62C8	PROC PrintB=*(BYTE n) [$A2$0]
$62CA	PROC PrintC=*(CARD n) [$20 CToStr$A5device]
$62CF	PROC PNum=*() [$50A2$5A0$20 Output$4C ChkErr]

```
P_PRINTB    LDX #$00        ; A2 00
P_PRINTC    JSR P_CTOSTR    ; 20 A8 62
            LDA DEVICE      ; A5 B7
P_PNUM      LDX #$50        ; A2 50
            LDY #$05        ; A0 05
            JSR P_OUTPUTQ   ; 20 0A 60
            JMP P_CHKERR    ; 4C CE 61
```

$62D9	PROC PrintBE=*(BYTE n) [$A2$0]
$62DB	PROC PrintCE=*(CARD n) [$20PrintC$4CPutE]

```
P_PRINTBE   LDX #$00        ; A2 00
P_PRINTCE   JSR P_PRINTC    ; 20 CA 62
            JMP P_PUTE      ; 4C 90 62
```

$62E1	PROC PrintBD=*(BYTE d, n) [$A0$0]
$62E3	PROC PrintCD=*(BYTE d, CARD n) [$A085$8A$A284$A2A6$20 CToStr$A0A5$4CPNum]

```
P_PRINTBD   LDY #$00        ; A0 00
P_PRINTCD   STA FRET        ; 85 A0
            TXA             ; 8A
            STY BPTR2       ; 84 A2
            LDX BPTR2       ; A6 A2
            JSR P_CTOSTR    ; 20 A8 62
            LDA FRET        ; A5 A0
            JMP P_PNUM      ; 4C CF 62
```

$62F2	PROC PrintBDE=*(BYTE d,n) [$A0$0]
$62F4	PROC PrintCDE=*(BYTE d,CARD n) [$20PrintCD$A0A5$4CPutDE]

```
P_PRINTBDE  LDY #$00        ; A0 00
P_PRINTCDE  JSR P_PRINTCD   ; 20 E3 62
            LDA FRET        ; A5 A0
            JMP P_PUTDE     ; 4C 9E 62
```

$62FC	PROC PrintI=*(INT n) [$A286$AA$A2A4$A5device]
$6303	PROC PrintID=*(BYTE d,INT n) [$C0$0$1610$48$A186$A284$2DA0$20PutD1$38$A9$0$A1E5AAA9$0$A2E5$A8$68$4CPrintCD]

```
P_PRINTI    STX BPTR2      ; 86 A2
            TAX            ; AA
            LDY BPTR2      ; A4 A2
            LDA DEVICE     ; A5 B7
P_PRINTID   CPY #$00       ; C0 00
            BPL L631D      ; 10 16
            PHA            ; 48
            STX FRET+1     ; 86 A1
            STY BPTR2      ; 84 A2
            LDY #$2D       ; A0 2D
            JSR P_PUTD1    ; 20 99 62
            SEC            ; 38
            LDA #$00       ; A9 00
            SBC FRET+1     ; E5 A1
            TAX            ; AA
            LDA #$00       ; A9 00
            SBC BPTR2      ; E5 A2
            TAY            ; A8
            PLA            ; 68
L631D       JMP P_PRINTCD  ; 4C E3 62
```

$6320	PROC PrintIE=*(INT n) [$20PrintI$4CPutE]

```
P_PRINTIE   JSR P_PRINTI   ; 20 FC 62
            JMP P_PUTE     ; 4C 90 62
```

$6326	PROC PrintIDE=*(BYTE d,INT n) [$20PrintID$A0A5$4CPutDE]

```
P_PRINTIDE  JSR P_PRINTID  ; 20 03 63
            LDA FRET       ; A5 A0
            JMP P_PUTDE    ; 4C 9E 62
```

$632E	PROC StrB=*(BYTE n, STRING s) [$A286$A384$A2$0$A2A4]
$6336	PROC StrC=*(CARD n, STRING s) [$A284$20 CToStr$C8$B9$550$A291$88$F810$60]
$6345	PROC StrI=*(INT n, STRING s) [$E0$0$ED10$A085$A186$A284$38$A9$0$A0E5$A8$A9$0$A1E5AA98$20CToStr$E8$8A$A8$B9$54F$A291$88$F8D0$8A$A291$C8$2DA9$A291$60]

```
P_STRB      STX BPTR2      ; 86 A2
            STY L00A3      ; 84 A3
            LDX #$00       ; A2 00
            LDY BPTR2      ; A4 A2
P_STRC      STY BPTR2      ; 84 A2
            JSR P_CTOSTR   ; 20 A8 62
            INY            ; C8
L633C       LDA L0550,Y    ; B9 50 05
            STA (BPTR2),Y  ; 91 A2
            DEY            ; 88
            BPL L633C      ; 10 F8
            RTS            ; 60
P_STRI      CPX #$00       ; E0 00
            BPL P_STRC     ; 10 ED
            STA FRET       ; 85 A0
            STX FRET+1     ; 86 A1
            STY BPTR2      ; 84 A2
            SEC            ; 38
            LDA #$00       ; A9 00
            SBC FRET       ; E5 A0
            TAY            ; A8
            LDA #$00       ; A9 00
            SBC FRET+1     ; E5 A1
            TAX            ; AA
            TYA            ; 98
            JSR P_CTOSTR   ; 20 A8 62
            INX            ; E8
            TXA            ; 8A
            TAY            ; A8
L6361       LDA L054F,Y    ; B9 4F 05
            STA (BPTR2),Y  ; 91 A2
            DEY            ; 88
            BNE L6361      ; D0 F8
            TXA            ; 8A
            STA (BPTR2),Y  ; 91 A2
            INY            ; C8
            LDA #$2D       ; A9 2D
            STA (BPTR2),Y  ; 91 A2
            RTS            ; 60
```

$6372	BYTE FUNC InputB=*() CARD FUNC InputC=*()
$6374	INT FUNC InputI=*() [$A5 device]
$6384	BYTE FUNC InputBD=*(BYTE d) CARD FUNC InputCD=*(BYTE d) INT FUNC InputID=*(BYTE d) [$13A2$8E$550$50A2$5A0$20InputD$50A9$5A2] BYTE FUNC ValB=*(STRING s) CARD FUNC ValC=*(STRING s) INT FUNC ValI=*(STRING s) [$A485$A586$A0$0$A084$A184$A284$A4B1$A385$A3E6$20A9$C8$A4D1$5D0$C8$A3C4$F730$A4B1$2DC9$3D0$A285$C8$A3C4$3610$A4B1$30C9$3030$3AC9$2C10$38$30E9$AA$A1A5$48$A0A5$A$A126$A$A126$18$A065$A085$68$A165$A185$A006$A126$18$8A$A065$A085$290$A1E6$C8$A3C4$CA30$A2A5$DF0$38$A9$0$A0E5$A085$A9$0$A1E5$A185$60]

```
F_INPUTBCI   LDA DEVICE     ; A5 B7
F_INPUTBCID  LDX #$13       ; A2 13
             STX L0550      ; 8E 50 05
             LDX # <L0550   ; A2 50
             LDY # >L0550   ; A0 05
             JSR P_INPUTD   ; 20 6B 62
             LDA # <L0550   ; A9 50
             LDX # >L0550   ; A2 05
F_VALBCI     STA L00A4      ; 85 A4
             STX L00A5      ; 86 A5
             LDY #$00       ; A0 00
             STY FRET       ; 84 A0
             STY FRET+1     ; 84 A1
             STY BPTR2      ; 84 A2
             LDA (L00A4),Y  ; B1 A4
             STA L00A3      ; 85 A3
             INC L00A3      ; E6 A3
             LDA #$20       ; A9 20
             INY            ; C8
L6399        CMP (L00A4),Y  ; D1 A4
             BNE L63A2      ; D0 05
             INY            ; C8
             CPY L00A3      ; C4 A3
             BMI L6399      ; 30 F7
L63A2        LDA (L00A4),Y  ; B1 A4
             CMP #$2D       ; C9 2D
             BNE L63AB      ; D0 03
             STA BPTR2      ; 85 A2
             INY            ; C8
L63AB        CPY L00A3      ; C4 A3
             BPL L63E5      ; 10 36
L63AF        LDA (L00A4),Y  ; B1 A4
             CMP #$30       ; C9 30
             BMI L63E5      ; 30 30
             CMP #$3A       ; C9 3A
             BPL L63E5      ; 10 2C
             SEC            ; 38
```

```
             SBC #$30       ; E9 30
             TAX            ; AA
             LDA FRET+1     ; A5 A1
             PHA            ; 48
             LDA FRET       ; A5 A0
             ASL A          ; 0A
             ROL FRET+1     ; 26 A1
             ASL A          ; 0A
             ROL FRET+1     ; 26 A1
             CLC            ; 18
             ADC FRET       ; 65 A0
             STA FRET       ; 85 A0
             PLA            ; 68
             ADC FRET+1     ; 65 A1
             STA FRET+1     ; 85 A1
             ASL FRET       ; 06 A0
             ROL FRET+1     ; 26 A1
             CLC            ; 18
             TXA            ; 8A
             ADC FRET       ; 65 A0
             STA FRET       ; 85 A0
             BCC L63E0      ; 90 02
             INC FRET+1     ; E6 A1
L63E0        INY            ; C8
             CPY L00A3      ; C4 A3
             BMI L63AF      ; 30 CA
L63E5        LDA BPTR2      ; A5 A2
             BEQ L63F6      ; F0 0D
             SEC            ; 38
             LDA #$00       ; A9 00
             SBC FRET       ; E5 A0
             STA FRET       ; 85 A0
             LDA #$00       ; A9 00
             SBC FRET+1     ; E5 A1
             STA FRET+1     ; 85 A1
L63F6        RTS            ; 60
```

Address	Declaration
$63F7	PROC PrintH=*(CARD n) [$A485$A586$4A9$A685$24A9$20Put$A9$0$4A2$A406$A526$2ACAF8D0$3069$3AC9$230$669$20Put$A6C6$E5D0$60]

```
P_PRINTH    STA L00A4       ; 85 A4
            STX L00A5       ; 86 A5
            LDA #$04        ; A9 04
            STA L00A6       ; 85 A6
            LDA #$24        ; A9 24
            JSR P_PUT       ; 20 92 62
L6404       LDA #$00        ; A9 00
            LDX #$04        ; A2 04
L6408       ASL L00A4       ; 06 A4
            ROL L00A5       ; 26 A5
            ROL A           ; 2A
            DEX             ; CA
            BNE L6408       ; D0 F8
            ADC #$30        ; 69 30
            CMP #$3A        ; C9 3A
            BMI L6418       ; 30 02
            ADC #$06        ; 69 06
L6418       JSR P_PUT       ; 20 92 62
            DEC L00A6       ; C6 A6
            BNE L6404       ; D0 E5
L641F       RTS             ; 60
```

Address	Declaration
$6420	PROC PrintF=*(STRING f, CARD a1,a2,a3,a4,a5) [$C085$C186$8C$5F0$A0$0$C0B1$C285$C2E6$DA2$A2B5$9D$5F0$CA$F8D0$8B86$8A86]
$643D	PROC PF2=*() [$8AE6$8AA4$C2C4$DAB0$C0B1$25C9$FD0$8AE6$C8$C0B1$25C9$6F0$45C9$8D0$9BA9$20Put$4CPF2$8BA4$8BE6$8BE6$A085$B9$5F0$BE$5F1$A0A4$43C0$E6F0$53C0$6D0$20Print$4CPF2$49C0$6D0$20PrintI$4CPF2$48C0$6D0$20PrintH$4CPF2$20PrintC$4CPF2]

```
P_PRINTF    STA L00C0       ; 85 C0
            STX L00C1       ; 86 C1
            STY L05F0       ; 8C F0 05
            LDY #$00        ; A0 00
            LDA (L00C0),Y   ; B1 C0
            STA L00C2       ; 85 C2
            INC L00C2       ; E6 C2
            LDX #$0D        ; A2 0D
L6431       LDA BPTR2,X     ; B5 A2
            STA L05F0,X     ; 9D F0 05
            DEX             ; CA
            BNE L6431       ; D0 F8
            STX L008B       ; 86 8B
            STX L008A       ; 86 8A
P_PF2       INC L008A       ; E6 8A
            LDY L008A       ; A4 8A
            CPY L00C2       ; C4 C2
            BCS L641F       ; B0 DA
            LDA (L00C0),Y   ; B1 C0
            CMP #$25        ; C9 25
            BNE L645A       ; D0 0F
            INC L008A       ; E6 8A
            INY             ; C8
            LDA (L00C0),Y   ; B1 C0
            CMP #$25        ; C9 25
            BEQ L645A       ; F0 06
            CMP #$45        ; C9 45
            BNE L6460       ; D0 08
            LDA #$9B        ; A9 9B
L645A       JSR P_PUT       ; 20 92 62
            JMP P_PF2       ; 4C 3D 64
L6460       LDY L008B       ; A4 8B
            INC L008B       ; E6 8B
            INC L008B       ; E6 8B
            STA FRET        ; 85 A0
            LDA L05F0,Y     ; B9 F0 05
            LDX L05F1,Y     ; BE F1 05
            LDY FRET        ; A4 A0
            CPY #$43        ; C0 43
            BEQ L645A       ; F0 E6
            CPY #$53        ; C0 53
            BNE L647E       ; D0 06
            JSR P_PRINT     ; 20 2F 62
            JMP P_PF2       ; 4C 3D 64
L647E       CPY #$49        ; C0 49
            BNE L6488       ; D0 06
            JSR P_PRINTI    ; 20 FC 62
            JMP P_PF2       ; 4C 3D 64
L6488       CPY #$48        ; C0 48
            BNE L6492       ; D0 06
            JSR P_PRINTH    ; 20 F7 63
            JMP P_PF2       ; 4C 3D 64
L6492       JSR P_PRINTC    ; 20 CA 62
            JMP P_PF2       ; 4C 3D 64
```

```
$6498   PROC Note=*(BYTE d,CARD POINTER s,BYTE POINTER o)
        [$A186$A284$A0A$A0A$AA$26A9$9D$342$20$E456$20
        ChkErr$A0$0$BD$34E$A391$BD$34C$A191$BD$34D$C8$A191$60]
```

```
P_NOTE      STX FRET+1      ; 86 A1
            STY BPTR2       ; 84 A2
            ASL A           ; 0A
            ASL A           ; 0A
            ASL A           ; 0A
            ASL A           ; 0A
            TAX             ; AA
            LDA #$26        ; A9 26
            STA ICCOM,X     ; 9D 42 03
            JSR CIOV        ; 20 56 E4
            JSR P_CHKERR    ; 20 CE 61
            LDY #$00        ; A0 00
            LDA ICAX5,X     ; BD 4E 03
            STA (L00A3),Y   ; 91 A3
            LDA ICAX3,X     ; BD 4C 03
            STA (FRET+1),Y  ; 91 A1
            LDA ICAX4,X     ; BD 4D 03
            INY             ; C8
            STA (FRET+1),Y  ; 91 A1
            RTS             ; 60
```

```
$64BF   PROC Point=*(BYTE d,CARD s,BYTE o)
        [$A186$A0A$A0A$98AA$9D$34D$A1A5$9D$34C$A3A5$9D$34E$25A9$9D$342$20$E
        456$4C ChkErr]
```

```
P_POINT     STX FRET+1      ; 86 A1
            ASL A           ; 0A
            ASL A           ; 0A
            ASL A           ; 0A
            ASL A           ; 0A
            TAX             ; AA
            TYA             ; 98
            STA ICAX4,X     ; 9D 4D 03
            LDA FRET+1      ; A5 A1
            STA ICAX3,X     ; 9D 4C 03
            LDA L00A3       ; A5 A3
            STA ICAX5,X     ; 9D 4E 03
            LDA #$25        ; A9 25
            STA ICCOM,X     ; 9D 42 03
            JSR CIOV        ; 20 56 E4
            JMP P_CHKERR    ; 4C CE 61
```

```
$64DF   MODULE ; GRAPHIC ROUTINES

$64E9   STRING dev_S="S:", dev_E="E:"

        PROC Graphics=*(BYTE m)
        [$48$A9$0$20 Close$CA9$A385$A9$0$AEdev_E$ACdev_E+1$20Open$6A9$20
        Close$68$A485$3029$1C49$A385$6A9$AEdev_S$ACdev_S+1$4COpen]
```

```
DEV_S       .BYTE $02,$53,$3A
L64E2       .BYTE $DF
L64E3       .BYTE $64
DEV_E       .BYTE $02,$45,$3A
L64E7       .BYTE $E4
L64E8       .BYTE $64
P_GRAPHICS  PHA             ; 48
            LDA #$00        ; A9 00
            JSR P_CLOSE     ; 20 29 62
            LDA #$0C        ; A9 0C
            STA L00A3       ; 85 A3
            LDA #$00        ; A9 00
            LDX L64E7       ; AE E7 64
            LDY L64E8       ; AC E8 64
            JSR P_OPEN      ; 20 F2 61
            LDA #$06        ; A9 06
            JSR P_CLOSE     ; 20 29 62
            PLA             ; 68
            STA L00A4       ; 85 A4
            AND #$30        ; 29 30
            EOR #$1C        ; 49 1C
            STA L00A3       ; 85 A3
            LDA #$06        ; A9 06
            LDX L64E2       ; AE E2 64
            LDY L64E3       ; AC E3 64
            JMP P_OPEN      ; 4C F2 61
```

```
$6517   PROC Position=*(CARD c,BYTE r)
        [$5B85$5C86$5A84]
$651D   PROC Pos1=*()
        [$5585$5686$5484$60]
```

```
P_POSITION  STA OLDCOL      ; 85 5B
            STX OLDCOL+1    ; 86 5C
            STY OLDROW      ; 84 5A
P_POS1      STA COLCRS      ; 85 55
            STX COLCRS+1    ; 86 56
            STY ROWCRS      ; 84 54
            RTS             ; 60
```

$6524	PROC GrIO=*() [$20Pos1$AD$2FD$8D$2FB$ADdev_S$A585$ADdev_S+1$A685$A9$0$A385$A485$6A9$60]	
P_GRIO	JSR P_POS1 ; 20 1D 65 LDA FILDAT ; AD FD 02 STA ATACHR ; 8D FB 02 LDA L64E2 ; AD E2 64 STA L00A5 ; 85 A5 LDA L64E3 ; AD E3 64	STA L00A6 ; 85 A6 LDA #$00 ; A9 00 STA L00A3 ; 85 A3 STA L00A4 ; 85 A4 LDA #$06 ; A9 06 RTS ; 60

$6540	PROC DrawTo=*(CARD c,BYTE r) [$20GrIO$11A0$4CXIO]	
P_DRAWTO	JSR P_GRIO ; 20 24 65 LDY #$11 ; A0 11 JMP P_XIO ; 4C A2 62	

$6548	BYTE FUNC Locate=*(CARD c,BYTE r) [$20Position$6A9$4CGetD]	
F_LOCATE	JSR P_POSITION ; 20 17 65 LDA #$06 ; A9 06 JMP F_GETD ; 4C 71 62	

$6550	PROC Plot=*(CARD c,BYTE r) [$20Pos1$6A9AE2FD$4CPutD]	
P_PLOT	JSR P_POS1 ; 20 1D 65 LDA #$06 ; A9 06 LDX FILDAT ; AE FD 02 JMP P_PUTD ; 4C 95 62	

$655B	PROC SetColor=*(BYTE reg,hue,lum) [$5C9$1610$A085$98$F29$A285$8A$A0A$A0A$A205$A0A6$9D$2C4$9D$D016$60]	
P_SETCOLOR	CMP #$05 ; C9 05 BPL L6575 ; 10 16 STA FRET ; 85 A0 TYA ; 98 AND #$0F ; 29 0F STA BPTR2 ; 85 A2 TXA ; 8A ASL A ; 0A	ASL A ; 0A ASL A ; 0A ASL A ; 0A ORA BPTR2 ; 05 A2 LDX FRET ; A6 A0 STA COLOR0,X ; 9D C4 02 STA COLPF0,X ; 9D 16 D0 L6575 RTS ; 60

$6576	PROC Fill=*(CARD c,BYTE r) [$20GrIO$12A0$4CXIO]	
P_FILL	JSR P_GRIO ; 20 24 65 LDY #$12 ; A0 12 JMP P_XIO ; 4C A2 62	

$657E	BYTE FUNC Rand=*(BYTE r) [AED20A$C9$0$9F0$8486$A2$0$8586$20MultI$A086$60]	
F_RAND	LDX RANDOM ; AE 0A D2 CMP #$00 ; C9 00 BEQ L658E ; F0 09 STX L0084 ; 86 84 LDX #$00 ; A2 00	STX L0085 ; 86 85 JSR P_MULTI ; 20 03 61 L658E STX FRET ; 86 A0 RTS ; 60

$6591	PROC Sound=*(BYTE v, p, d, vol) [AA284$A8$7C9$530$64A0$20 Error$998A$D200$A2A5$A0A$A0A$A305$99$D201$60]

```
P_SOUND     ASL A           ; 0A
            STY BPTR2       ; 84 A2
            TAY             ; A8
            CMP #$07        ; C9 07
            BMI L659E       ; 30 05
            LDY #$64        ; A0 64
            JSR P_ERROR()   ; 20 73 60
L659E       TXA             ; 8A
            STA AUDF1,Y     ; 99 00 D2
            LDA BPTR2       ; A5 A2
            ASL A           ; 0A
            ASL A           ; 0A
            ASL A           ; 0A
            ASL A           ; 0A
            ORA L00A3       ; 05 A3
            STA AUDC1,Y     ; 99 01 D2
            RTS             ; 60
```

$65AE	PROC SndRst=*() [AD232$EF29$8D$232$8D$D20F$A9$0$8A2$9D$D200CAFA10$60]

```
P_SNDRST    LDA SSKCTL      ; AD 32 02
            AND #$EF        ; 29 EF
            STA SSKCTL      ; 8D 32 02
            STA SKCTL       ; 8D 0F D2
            LDA #$00        ; A9 00
            LDX #$08        ; A2 08
L65BD       STA AUDF1,X     ; 9D 00 D2
            DEX             ; CA
            BPL L65BD       ; 10 FA
            RTS             ; 60
```

$65C4	BYTE FUNC Paddle=*(BYTE p) [$BDAA$270$A085$60]

```
F_PADDLE    TAX             ; AA
            LDA PADDL0,X    ; BD 70 02
            STA FRET        ; 85 A0
            RTS             ; 60
```

$65CB	BYTE FUNC PTrig=*(BYTE p) [$A2$0$4C9$330$E8$329$A8$BD$D300$39*+5$A085$60$804$8040]

```
F_PTRIG     LDX #$00        ; A2 00
            CMP #$04        ; C9 04
            BMI L65D4       ; 30 03
            INX             ; E8
            AND #$03        ; 29 03
L65D4       TAY             ; A8
            LDA PORTA,X     ; BD 00 D3
            AND L65DE,Y     ; 39 DE 65
            STA FRET        ; 85 A0
            RTS             ; 60
L65DE       .BYTE $04,$08,$40,$80
```

$65E2	BYTE FUNC Stick=*(BYTE p) [$A2$0$2C9$330$E8$129$BDA8$D300$88$4D0$4A4A$4A4A$F29$A085$60]

```
F_STICK     LDX #$00        ; A2 00
            CMP #$02        ; C9 02
            BMI L65EB       ; 30 03
            INX             ; E8
            AND #$01        ; 29 01
L65EB       TAY             ; A8
            LDA PORTA,X     ; BD 00 D3
            DEY             ; 88
            BNE L65F6       ; D0 04
            LSR A           ; 4A
            LSR A           ; 4A
            LSR A           ; 4A
            LSR A           ; 4A
L65F6       AND #$0F        ; 29 0F
            STA FRET        ; 85 A0
            RTS             ; 60
```

$65FB	BYTE FUNC STrig=*(BYTE p) [$BDAA$D010$A085$60]

```
F_STRIG     TAX             ; AA
            LDA TRIG0,X     ; BD 10 D0
            STA FRET        ; 85 A0
            RTS             ; 60
```

$6602	BYTE FUNC Peek=*(CARD a) CARD FUNC PeekC=*(CARD a) [$A285$A386$A0$0$A2B1$A085$C8$A2B1$A185$60]

```
F_PEEKC     STA BPTR2       ; 85 A2
            STX L00A3       ; 86 A3
            LDY #$00        ; A0 00
            LDA (BPTR2),Y   ; B1 A2
            STA FRET        ; 85 A0
            INY             ; C8
            LDA (BPTR2),Y   ; B1 A2
            STA FRET+1      ; 85 A1
            RTS             ; 60
```

$6612	PROC Poke=*(CARD a,BYTE v) [$A085$A186$A098$9100$60A0]

```
P_POKE      STA FRET        ; 85 A0
            STX FRET+1      ; 86 A1
            TYA             ; 98
            LDY #$00        ; A0 00
            STA (FRET),Y    ; 91 A0
            RTS             ; 60
```

$661C	PROC PokeC=*(CARD a,v) [$20Poke$A5C8$91A3$60A0]

```
P_POKEC     JSR P_POKE      ; 20 12 66
            INY             ; C8
            LDA L00A3       ; A5 A3
            STA (FRET),Y    ; 91 A0
            RTS             ; 60
```

$6625 $662B	PROC Zero=*(BYTE POINTER a,CARD s) [48A9$0$A485$68] PROC SetBlock=*(BYTE POINTER a,CARD s,BYTE v) [$A085$A186$A284$A0$0$A4A5$A3A6$10F0$A091$C8$FBD0$A1E6$A3C6$F5D0$3F0$A091$C8$A2C4$F9D0$60]

```
P_ZERO      PHA             ; 48
            LDA #$00        ; A9 00
            STA L00A4       ; 85 A4
            PLA             ; 68
P_SETBLOCK  STA FRET        ; 85 A0
            STX FRET+1      ; 86 A1
            STY BPTR2       ; 84 A2
            LDY #$00        ; A0 00
            LDA L00A4       ; A5 A4
            LDX L00A3       ; A6 A3
            BEQ L6649       ; F0 10
L6639       STA (FRET),Y    ; 91 A0
            INY             ; C8
            BNE L6639       ; D0 FB
            INC FRET+1      ; E6 A1
            DEC L00A3       ; C6 A3
            BNE L6639       ; D0 F5
            BEQ L6649       ; F0 03
L6646       STA (FRET),Y    ; 91 A0
            INY             ; C8
L6649       CPY BPTR2       ; C4 A2
            BNE L6646       ; D0 F9
            RTS             ; 60
```

$664E	PROC MoveBlock=*(BYTE POINTER d,s,CARD sz) [$A085$A186$A284$A0$0$A5A5$16F0$A2B1$A091$C8$F9D0$A1E6$A3E6$A5C6$F1D0$5F0$A2B1$A091$C8$A4C4$F7D0$60]

```
P_MOVEBLOCK STA FRET        ; 85 A0
            STX FRET+1      ; 86 A1
            STY BPTR2       ; 84 A2
            LDY #$00        ; A0 00
            LDA L00A5       ; A5 A5
            BEQ L6670       ; F0 16
L665A       LDA (BPTR2),Y   ; B1 A2
            STA (FRET),Y    ; 91 A0
            INY             ; C8
            BNE L665A       ; D0 F9
            INC FRET+1      ; E6 A1
            INC L00A3       ; E6 A3
            DEC L00A5       ; C6 A5
            BNE L665A       ; D0 F1
            BEQ L6670       ; F0 05
L666B       LDA (BPTR2),Y   ; B1 A2
            STA (FRET),Y    ; 91 A0
            INY             ; C8
L6670       CPY L00A4       ; C4 A4
            BNE L666B       ; D0 F7
            RTS             ; 60
```

$6675	INT FUNC SCompare=*(STRING a,b) [$A485$A586$A284$A0$0$A084$A184$A4B1$A2D1$3F0$20*+21$C9$0$1D0$60$A685$C8$A4B1$A2D1$5D0$A6C4$F590$60$FFA2$A086$390$A2B1$E8$A186$60]

```
F_SCOMPARE  STA L00A4       ; 85 A4
            STX L00A5       ; 86 A5
            STY BPTR2       ; 84 A2
            LDY #$00        ; A0 00
            STY FRET        ; 84 A0
            STY FRET+1      ; 84 A1
            LDA (L00A4),Y   ; B1 A4
            CMP (BPTR2),Y   ; D1 A2
            BEQ L668A       ; F0 03
            JSR L669D       ; 20 9D 66
L668A       CMP #$00        ; C9 00
            BNE L668F       ; D0 01
            RTS             ; 60
L668F       STA L00A6       ; 85 A6
L6691       INY             ; C8
            LDA (L00A4),Y   ; B1 A4
            CMP (BPTR2),Y   ; D1 A2
            BNE L669D       ; D0 05
            CPY L00A6       ; C4 A6
            BCC L6691       ; 90 F5
            RTS             ; 60
L669D       LDX #$FF        ; A2 FF
            STX FRET        ; 86 A0
            BCC L66A6       ; 90 03
            LDA (BPTR2),Y   ; B1 A2
            INX             ; E8
L66A6       STX FRET+1      ; 86 A1
            RTS             ; 60
```

$66A9	PROC SCopy=*(STRING d,s) [$A085$A186$A284$A0$0$A2B1$A091$8F0$A8$A2B1$A091$88$F9D0$60]

```
P_SCOPY     STA FRET        ; 85 A0
            STX FRET+1      ; 86 A1
            STY BPTR2       ; 84 A2
            LDY #$00        ; A0 00
            LDA (BPTR2),Y   ; B1 A2
L66B3       STA (FRET),Y    ; 91 A0
            BEQ L66BF       ; F0 08
L66B7       TAY             ; A8
L66B8       LDA (BPTR2),Y   ; B1 A2
            STA (FRET),Y    ; 91 A0
            DEY             ; 88
            BNE L66B8       ; D0 F9
L66BF       RTS             ; 60
```

$66C0	PROC SCopyS=*(STRING d,s,BYTE b,e) [$A085$A186$A284$A0$0$A2B1$A5C5$2B0$A585$A4C6$18$A2A5$A465$A285$290$A3E6$38$A5A5$A4E5$2B0$A9$0$4CSCopy+10]

```
P_SCOPYS    STA FRET        ; 85 A0
            STX FRET+1      ; 86 A1
            STY BPTR2       ; 84 A2
            LDY #$00        ; A0 00
            LDA (BPTR2),Y   ; B1 A2
            CMP L00A5       ; C5 A5
            BCS L66D0       ; B0 02
            STA L00A5       ; 85 A5
L66D0       DEC L00A4       ; C6 A4
            CLC             ; 18
            LDA BPTR2       ; A5 A2
            ADC L00A4       ; 65 A4
            STA BPTR2       ; 85 A2
            BCC L66DD       ; 90 02
            INC L00A3       ; E6 A3
L66DD       SEC             ; 38
            LDA L00A5       ; A5 A5
            SBC L00A4       ; E5 A4
            BCS L66E6       ; B0 02
            LDA #$00        ; A9 00
L66E6       JMP L66B3       ; 4C B3 66
```

<table>
<tr><td>$66E9</td><td colspan="2">PROC SAssign=*(STRING d,s,BYTE b,e)
[$A085$A186$A284$A0$0$A2B1$DF0$A685$A4C6$38$A5A5$A4E5$2F0$1B0$AA60$A6C5$890$18$A6A5$AA$A465$A585$A5A5$A0D1$390$A091$18$A0A5$A465$A085$290$A1E6$4C8ASCopy+14]</td></tr>
<tr><td colspan="2">

```
P_SASSIGN   STA FRET        ; 85 A0
            STX FRET+1      ; 86 A1
            STY BPTR2       ; 84 A2
            LDY #$00        ; A0 00
            LDA (BPTR2),Y   ; B1 A2
            BEQ L6702       ; F0 0D
            STA L00A6       ; 85 A6
            DEC L00A4       ; C6 A4
            SEC             ; 38
            LDA L00A5       ; A5 A5
            SBC L00A4       ; E5 A4
            BEQ L6702       ; F0 02
            BCS L6703       ; B0 01
L6702       RTS             ; 60
L6703       TAX             ; AA
            CMP L00A6       ; C5 A6
            BCC L6710       ; 90 08
            CLC             ; 18
```

</td><td>

```
            LDA L00A6       ; A5 A6
            TAX             ; AA
            ADC L00A4       ; 65 A4
            STA L00A5       ; 85 A5
L6710       LDA L00A5       ; A5 A5
            CMP (FRET),Y    ; D1 A0
            BCC L6719       ; 90 03
            STA (FRET),Y    ; 91 A0
            CLC             ; 18
L6719       LDA FRET        ; A5 A0
            ADC L00A4       ; 65 A4
            STA FRET        ; 85 A0
            BCC L6723       ; 90 02
            INC FRET+1      ; E6 A1
L6723       TXA             ; 8A
            JMP L66B7       ; 4C B7 66
            RTS             ; 60
```

</td></tr>
<tr><td></td><td colspan="2">MODULE ; for user</td></tr>
<tr><td colspan="2">;</td><td></td></tr>
</table>

K - Tipps, Tricks & Anregungen

Allgemeine Informationen

Die Informationen hier passen aufgrund ihrer Thematik nicht in die anderen Teile des Handbuchs. Es geht dabei um Fehlerquellen, Programmiertipps und anderes mehr.

K.1 Umgang mit dem Modul

Action! scheint einige Eigenarten zu besitzen, deren Ursachen nicht immer klar oder bekannt sind. Das hier gesammelte Wissen soll helfen, Probleme zu vermeiden.

Erste Editorzeile

Wenn man nach einem Kaltstart erstmalig in den Editor geht, muss man 2x <RETURN> drücken, um aus der obersten Zeile herauszukommen. Möglicherweise wird da etwas nicht richtig initialisiert. Deshalb die oberste Zeile im Programmtext immer leer lassen.

Zeilenlänge im Editor

Die im Optionsmenü des Monitors maximal einstellbare Zeilenlänge von 240 Zeichen kann ggf. zu Fehlern führen. Der Editor kann dann z.B. das Anzeigefenster nicht mehr bis an den rechten Rand verschieben und/oder stürzt ab. Eine sichere Grenze scheint bei maximal 160 Zeichen pro Zeile zu liegen.

Zurück nach DOS

Manchmal kommt es vor, dass man im Monitor nach Eingabe von 'D<RETUIRN>' in der Kommandozeile des Monitors hängen bleibt. Falls das System nicht abgestürzt ist, gelangt man beim nächsten Versuch zurück zum DOS.

Tastaturwiederholfunktion

Action! verwendet einen eigenen Tastaturbeschleuniger. Arbeitet der eingesetzte A8 bereits mit verkürzter Ansprechzeit (729;$2D9) und/oder erhöhter Geschwindigkeit (730;$2DA), wird das System durch die zusätzliche Beschleunigung der Tastaturfunktionen eventuell nicht mehr bedienbar.

Daher vor dem Sprung ins Modul die Standardwerte ($2D9=$28, $2DA=$5) hineinschreiben, die Software zur Tastaturbeschleunigung abschalten oder ein Standard-OS verwenden. Anwender von SpartaDOS X 4.4x umgehen das Problem durch Einschalten des Tastaturpuffers (KEY ON), der extra für Action! angepasst wurde.

Bildschirmbeschleuniger

Action! verfügt über eine beschleunigte Bildschirmausgabe. Falls bereits andere Beschleuniger im System aktiv sind, kann es sein, dass Action! nicht oder nicht richtig funktioniert. Sollte es Probleme geben, den Beschleuniger abschalten. Ist er in das OS integriert, auf das Standard-OS wechseln.

K.2 Stolperfallen in Action!

Wie in jeder Programmierumgebung gibt es auch hier einige Dinge, die man schlicht wissen und beachten muss. Diese werden nachfolgend beschrieben.

Laden eines kompilierten Programms

Beim Laden eines kompilierten Programms, egal ob mit oder ohne Modul, gibt es manchmal überraschende Ergebnisse.

So starten manche DOS ein geladenes Programm einfach an der ersten geladenen Adresse. Durch Einfügen der Zeile

```
BYTE RTS=[$60]
```

als allererste Zeile im Programmtext wird dieses Problem gelöst.

Außerdem speichert Action! das kompilierte Programm nur mit einer INIT-Adresse ab. Das ist insofern problematisch, weil nach dem Ende der INIT-Routine eine RUN-Adresse gesucht wird, die aber nicht existiert.

Diese beiden Ursachen können dafür verantwortlich sein, dass ein Programm gar nicht, mehrmals oder auch einfach korrekt ausgeführt wird, je nach verwendetem DOS.

Wird SpartaDOS X V. 4.47 als DOS eingesetzt, werden diese beiden Probleme und andere Fälle durch die speziellen Lademodi der Kommandos CAR und LOAD behoben.

Die fehlende RUN-Adresse lässt sich auch mithilfe anderer Tools ergänzen, wie z.B. dem Superpacker.

RUN-Adresse anhängen
Am Beispiel der PROC Hallo aus Kapitel I wird aufgezeigt, wie eine RUN-Adresse angehängt werden kann.

Zuerst den Programmtext wie folgt ergänzen:

```
(leere erste Zeile)
MODULE

SET $000E=$4000
SET $0491=$4000

BYTE RTS=[$60]

PROC Hallo()
  PRINTE("Hallo Leute!")
RETURN
```

Als Nächstes kompilieren und als HALLO.BIN abspeichern. Nun von DOS aus den Superpacker[125] laden und

- das kompilierte File HALLO.BIN in den Superpacker laden,
- am Ende eine RUN-Adresse einfügen ($2E0-$2E1),
- den Wert der RUN-Adresse auf den der INIT-Adresse($2E2-$2E3) einstellen,
- die Zeile mit der INIT-Adresse entfernen,
- das File als HELLO.COM abspeichern.

Eine ganz andere Lösung desselben Problems ist das Patchen des Action!-Moduls. Im ACTION! ROM stehen ab

```
$3EC1 .WORD $02E2
      .WORD $02E3
```

125) BeweDOS Lieferumgang.

Beim Kompilieren verwendet ACTION! diese 4 Bytes, um an das Ende des COM-Files den INIT-Vektor anzuhängen.

INIT sollte aber nur dann verwendet werden, wenn noch weiterer Code nachgeladen werden soll. Der letzte Teil sollte dann eine RUN Adresse haben. Werden nur Action!-Programme geschrieben, die nichts nachladen, wäre es praktisch, gleich eine RUN-Adresse am Ende der Datei zu haben.

Das Patchen des ROMs im Modul auf

```
$3EC1 .WORD $02E0
      .WORD $02E1
```

sorgt genau dafür.

Aufrufe von Prozeduren/Funktionen mit mehr als 3 Byte Parametern
Programme, die mehr als 3 Byte Parameter verwenden, benötigen entweder das Modul oder die Runtime. Dies liegt daran, dass beim Aufruf einer PROC oder FUNC mit mehr als 3 Bytes an Parametern die Werte in den Speicherstellen $A3 bis $AF gespeichert werden. Die so aufgerufene PROC oder FUNC verwendet dann eine Routine aus dem Modul oder der Runtime. In der Runtime heißt die Routine SARGS.

Offsets per TYPE Deklaration
Die Verwendung einer TYPE-Deklaration erzeugt einen zufälligen Fehler, sobald der Code-Offset (Inhalt der Speicherstelle $B5) nicht null ist. Der Fehler wird höchstwahrscheinlich erst bei Benutzung der Laufzeitbibliothek auffallen. Deshalb alle TYPE-Deklarationen vornehmen, bevor man den Code-Offset ändert.

Beispiel:

```
; Programmanfang --
; Zuerst die TYPEs deklarieren
TYPE IOCB = [ BYTE Id, Devnum,
              Command, Status ]
; Danach, falls gewuenscht,
; den Offset aendern
SET $B5 = $1000
; Beispiel: offset=4096
```

Offsets
Ein Code-Offset größer als $7FFF (e.g. ein negativer Offset, falls man den als Typ INT betrachtet) lässt den Compiler unzulässigen Code generieren. Das wirkt sich besonders bei Verwendung der Laufzeitbibliothek aus.

Eine direkte Lösung gibt es leider nicht. Aber man kann das Programm zum Verschieben nutzen, das später in diesem Anhang thematisiert wird. Dieses lässt sich auch in Assembler verwenden.

ATARI DOS
Wird Action! unter ATARI DOS und dessen Derivaten beendet, stürzt es ab, wenn das 'DUP.SYS' auf dem Laufwerk #1 fehlt.

ARRAYs und ELSEIF
Es gibt offenbar einen ziemlich obskuren Fehler in Action!, der bei Verwendung von ELSEIF auftritt. Betroffen sind besonders Anweisungen in der Form

```
ELSEIF a(i) = 0 THEN ...
```

wobei 'a' ein ARRAY ist und 'i' eine CARD oder INT ist.

Oder Anweisungen wie

```
ELSEIF p^ = 0 THEN ...
```

wobei 'p' ein Pointer ist.
Es wird dann fehlerhafter Code erzeugt.

Abhilfe: Gibt es nicht. Das Problem scheint sich am besten umschiffen zu lassen durch Programmierungen wie

```
t = a(i)    ; t is an INTEGER
...
ELSEIF t=0 THEN ...
```

Das funktioniert.

TYPE Pointer Argumente
Die Deklaration von PROCs oder FUNCs mithilfe von Record Pointer Argumenten wird mit Ausnahme des ersten Arguments nicht korrekt kompiliert. Beispielsweise erzeugt dieser Programmcode einen Fehler 7 (ungültige Argumentenliste):

```
TYPE REC=[...]
    ...
PROC Test( BYTE x, REC POINTER p )
```

Abhilfe: weglassen des Kommas in der Argumentenliste für die PROC/FUNC, wie in:

```
PROC Test( BYTE x
           REC POINTER p )
```

Dies ist nur eine Notlösung und funktioniert möglicherweise nicht in zukünftigen Versionen, aber die korrekte Deklaration mit dem Komma schon.

Anmerkung: Bei mir in der Version 3.6 scheint der Fehler nicht aufzutreten. Bitte um Hinweise, wenn jemand andere Informationen herausfindet.

Fehler in der Anleitung zum 'Runtime Package'
Der auf Seite 17 der originalen Anleitung gezeigte Code 'DEFINE for ROM' verursacht Fehler, wenn lokale Variablen verwendet werden. Diese Form der Definition funktioniert:

```
     DEFINE ROM = "BYTE ZZQQJUNK
                  -------------
                  SET $680 = $E^
                  SET $B5  = $5800
                  SET $E   = $682^"
```

K.3 Tipps zu "Temps"
Der Magazinartikel "Lights, Camera, Action!" von Dave Plotkin, der in der ANTIC vom Juli 1984 erschien, befasst sich mit Routinen, die das Schreiben von Interrupt-Handlern in Action! möglich machen.

Mit dem Artikel erschienen zwei Routinen (genauer DEFINEs) namens "SaveTemps" und "GetTemps". Diese Routinen ließen sich nur verwenden, wenn in der Interrupt-Routine keine Berechnungen außer Addition und Subtraktion stattfinden. Die nachfolgenden Versionen dieser beiden Routinen funktionieren generell einwandfrei.

Diese DEFINEs müssen im eigenen Programm vor der Deklaration der Interrupt-Routine eingefügt werden. Die Kommentare können dabei weggelassen werden, sie dienen nur zur Erläuterung.

```
DEFINE SaveTemps=
  "[
     $A2 $07     ;       LDX #7
     $B5 $C0     ; LOOP  LDA $C0,X
     $48         ;       PHA
     $B5 $A0     ;       LDA $A0,X
     $48         ;       PHA
     $B5 $80     ;       LDA $80,X
     $48         ;       PHA
     $B5 $A8     ;       LDA $A8,X
     $48         ;       PHA
     $CA         ;       DEX
     $10 $F1     ;       BPL LOOP
     $A5 $D3     ;       LDA $D3
     $48         ;       PHA
  ]"
DEFINE GetTemps=
  "[
     $68         ;       PLA
     $85 $D3     ;       STA $D3
     $A2 $00     ;       LDX #0
     $68         ; LOOP  PLA
     $95 $A8     ;       STA $A8,X
     $68         ;       PLA
     $95 $80     ;       STA $80,X
     $68         ;       PLA
     $95 $A0     ;       STA $A0,X
```

```
     $68          ;       PLA
     $95 $C0      ;       STA $C0,X
     $E8          ;       INX
     $E0 $08      ;       CPX #8
     $D0 $EF      ;       BNE LOOP
  ]"
```

Diese beiden Routinen müssen in der eigenen Interrupt-Routine wie folgt eingesetzt werden:

```
   ; Die eigene Interrupt-Routine.
   PROC InterruptRoutine()
   ; Lokale Deklarationen, falls noetig.
      BYTE a, b, c, etc.
      ; Erste Code-Zeile der
      ; Prozedur SaveTemps

      ; Eigene Interrupt-Routine
      ; folgt hier.

      GetTemps          ; Letzte Code-Zeile der
                        ; der Prozedur.
   [$6C OldVBI]         ; Eine besondere Art
                        ; einen VBI zu beenden
                        ; folgt hiernach.
```

Nachfolgend ein Beispielprogramm dazu.

Dieses Programm richtet die Routine 'ChangeColor' als "VBI-Routine ein. Zum Beenden des Programms die <START>-Taste drücken.

```
DEFINE SaveTemps=
  "[ $A2 $07 $B5 $C0 $48
     $B5 $A0 $48 $B5 $80
     $48 $B5 $A8 $48 $CA
     $10 $F1 $A5 $D3 $48 ]"

DEFINE GetTemps=
  "[ $68 $85 $D3 $A2 $00 $68
     $95 $A8 $68 $95 $80 $68
     $95 $A0 $68 $95 $C0 $E8
     $E0 $08 $D0 $EF ]"

CARD OldVBI    ; speichert den
               ; bisherigen
               ; Inhalt des
               ; VBI-Vektors.

; Diese Prozedur aendert die Hinter-
; grundfarbe mit Zufallswerten.
; Die Hauptroutine sorgt dafuer, dass
```

```
; der Code waehrend des 'Deferred'
; VBI ausgefuehrt wird.
   PROC ChangeColor()
      BYTE hue, lum

      SaveTemps
      hue = Rand( 16 )
      lum = Rand( 16 )
      SetColor(2,hue,lum)
      GetTemps

   [ $6C OldVBI ]  ; VBIs muessen so beendet
                   ; werden. $6C ist ein
                   ; indirekter
                   ; Sprungbefehl
                   ; für den 6502-Prozessor.

   PROC Test()     ; Hauptroutine
      BYTE critic=$42,     ; Critical I/O Flag
           console=$D01F   ; Hardwareregister
                           ; der Konsolentasten
      CARD VBIvec=$224     ; Deferred VBI Vektor
      ; Die VBI-Routine muss so
      ; eingerichtet werden:
      critic = 1
      OldVBI = VBIvec
      VBIvec = ChangeColor
      critic = 0
      ; ChangeColor laeuft jetzt
      ; als VBI-Routine. Da der
      ; Code hier keine weiteren
      ; Aufgaben hat, folgt
      ; eine Warteschleife
      ; bis die START-Taste
      ; gedrueckt wird.
      WHILE console&1
      DO
      OD

      ; Nun VBI-Routine abschalten.
      critic = 1
      VBIvec = OldVBI
      critic = 0
   RETURN
```

Diese Methode zur Zwischenspeicherung und Wiederherstellung von Action! Zero-Page-Variablen kann auch für das Schreiben EINFACHER Maschinenspracheroutinen in Action! verwendet werden. Die Hauptroutine sollte dann 'SaveTemps' als erste und 'GetTemps' als letzte auszuführende Programmzeile vor 'RETURN' aufweisen.

K.4 Action! Object Code Relocation Kit

Mit den nachfolgenden Programmen von John DeMar kann ein Action!-Programm an eine andere als die ursprünglich kompilierte Adresse geladen und gestartet werden. Das funktioniert auch mit Objektdateien in Assembler, vorausgesetzt die Anweisungen dazu werden befolgt.

Aus zwei Objektdateien als Eingabe und wird eine dritte Datei erzeugt, die dann an die gewünschte Adresse geladen und gestartet werden kann.

Relocation Generator und Runtime Relocator

Diese Programme generieren aus einem kompilierten Action!- oder Assemblerprogramm eine Objektdatei, die automatisch umadressiert wird. Die ursprüngliche Objektdatei muss ein einstufiges Boot-Programm mit nur einer Ursprungsadresse sein; mit Ausnahme einer nachfolgenden Run- oder Init-Adresse. Die nachfolgenden Anweisungen beschreiben die erforderlichen Schritte, um eine entsprechend geänderte Objektdatei zu erzeugen. Diese neue Objektdatei kann dann an andere binäre Ladedateien angehängt werden bzw. man darf andere binäre Ladedateien daran anhängen. Der 'Relocation Generator' RELGEN wird an die nächstmögliche Speicherseitengrenze nach MEMLO geladen. Da zwei Versionen einer Objektdatei verglichen werden, bietet es sich an, alle Variablen auf null zu setzen, um die Umadressierungstabelle möglichst klein zu halten. Zufällige Daten in nicht initialisierten Variablen könnten als Maschinencode interpretiert werden, der dann einer Umadressierung bedarf.

Schritt 1

Der Programmcode wird in einen passenden Speicherbereich kompiliert[126] (oder assembliert), der nicht mit dem DOS in Konflikt steht.

Schritt 2

Der Code wird erneut kompiliert, aber an eine um $100 höhere Adresse als beim ersten Mal.

Schritt 3

Starten des Programms RELGEN.ACT aus dem Monitor heraus. Es wird als Eingabe die zuvor kompilierten Objektdateien erfragen. Nicht vergessen, dabei den Zusatz 'Dn:' für das betreffende Laufwerk zum Dateinamen anzugeben. Das Programm vergleicht die beiden Objektdateien und speichert die Unterschiede als Offsets in die Datei. Diese Information wird im Action!-Format in einer Datei gespeichert, die den ursprünglichen Namen hat, aber den Extender '.GEN' aufweist. Diese Datei wird im nächsten Schritt verwendet. Außerdem erzeugt das Programm ein Abbild der ursprünglichen Objektdatei, allerdings mit der Ursprungsadresse null. Diese Datei hat den Extender '.REL' und erleichtert den Umadressierungsprozess. Es wird in Schritt 5 verwendet.

Anmerkung: RELGEN.ACT benötigt 4 gleichzeitig geöffnete DOS-Dateien. ATARI-DOS ist auf 3,. SpartaDOS X auf 5 Puffer voreingestellt. Mit dem Monitorkommando

```
SET $709=4
```

stellt man das ein und kann dann ins DOS gehen, dieses auf Disk schreiben und dann erneut booten. Nun weist ATARI-DOS vier gleichzeitig geöffnete Dateien auf.

126) Kapitel V, Abschnitt 2 für das Kompilieren an eine bestimmte Adresse.

Schritt 4
Nun das Programm RELOC.ACT in den Editor laden. Es handelt sich dabei um einen einfachen 'Runtime Relocator'. Die mit dem Programm RELGEN.ACT erstellte Datei (mit dem Extender '.GEN') muss nun mithilfe des Editors durch Einlesen in die Datei RELOC.ACT integriert werden. Dazu wird der Cursor an die vorgegebene Position gesetzt und dann eingelesen. Danach kann kompiliert werden, wobei aber mit SET sicherzustellen ist, dass in einen Bereich oberhalb des zu erwartenden Programmendes kompiliert wird. Der erzeugte Objektcode muss nun gespeichert werden, dann kann man ins DOS gehen.

Schritt 5
Mit dem Kommando COPY wird nun die mit RELGEN.ACT erzeugte '.REL'-Datei an die im Schritt 4 mit RELOC.ACT zusammengesetzte 'Relocator'-Datei angehängt. Für ATARI-DOS sieht das wie folgt aus:

```
C
Copy from,to:
TEST.REL,AUTORUN.SYS/A
```

Hier wird angenommen, dass die in Schritt 4 erstellte Datei AUTORUN.SYS genannt wurde.

Schritt 6
Nun endlich kann die so erzeugte Ergebnisdatei entweder von DOS aus geladen oder als dauerhafte Anwendung genutzt werden, sofern sie als AUTORUN.SYS gespeichert wurde.

RELGEN.ACT
```
MODULE ;RELGEN.ACT

;COPYRIGHT 1984, QMI, JS DeMar
;REV. 1.1, March 20, 1984
;VERSION II, August, 1984

;OBJECT CODE RELOCATION GENERATOR for
;Action! compiled binary-load files.

DEFINE in1="1",
       in2="2",
       out1="3",
       out2="4"

BYTE abrt

PROC MyError(BYTE a,x,y)

  IF y=170 THEN
    PrintE("ERROR File not found!")
  ELSE
    Print("ERROR! ")
    PrintBE(y)
```

```
  FI
  abrt=1
RETURN

PROC Ferror()

  BYTE t,clock=$14

  PrintE("ERROR in Output filespec!")
  t=clock-$80
  DO
    UNTIL t=clock
  OD
RETURN

PROC EndIt()

  Close(in1)
  Close(in2)
  Close(out1)
  Close(out2)
RETURN

PROC Main()

  CARD start1,start2,end1,end2
  CARD offsets,offsete,i,count,hits
  CARD test1,test2,old1,old2,old3,old0
  BYTE x,z,j,wnum,d1,d2,
       sthigh

  BYTE ARRAY fname1(18),fname2(18),
             fnameout1(18),fnameout2(18)

  DO
    PrintE("}    Relocation Code Generator II ")
    PrintE("#           JS DeMar, 8/84 ")
    PutE()
    PrintE("  Requires two code files compiled")
    PrintE("      with an offset of $0100.")
    PutE()

    Print("Filespec for code A >")
    InputMD(device,fname1,18)
    PutE()
    Print("Filespec for code B >")
    InputMD(device,fname2,18)
    PutE()

    Scopy(fnameout1,fname1)
    SCopy(fnameout2,fnameout1)
```

```
    j=1
    IF fnameout1(1)#'D
      OR fnameout1(0)<4 THEN
      Ferror()
    ELSEIF fnameout1(2)=': THEN
      z=0
    ELSEIF fnameout1(3)=': THEN
      z=1
    FI
  DO
  x=fnameout1(j)
  j==+1
  IF x=$20 THEN
    EXIT
  ELSEIF x='. THEN
    EXIT
  ELSEIF j>fnameout1(0) THEN
    j==+1
    EXIT
  ELSEIF j>11+z THEN
    Ferror()
  FI
OD

fnameout1(j-1)='.
fnameout1(j)='G
fnameout1(j+1)='E
fnameout1(j+2)='N
fnameout1(0)=j+2

j=1
IF fnameout2(1)#'D
      OR fnameout2(0)<4 THEN
  Ferror()
ELSEIF fnameout2(2)=': THEN
  z=0
  EXIT
ELSEIF fnameout2(3)=': THEN
  z=1
  EXIT
FI
OD
DO
  x=fnameout2(j)
  j==+1
  IF x=$20 THEN
    EXIT
  ELSEIF x='. THEN
    EXIT
  ELSEIF j>fnameout2(0) THEN
    j==+1
```

```
    EXIT
  ELSEIF j>11+z THEN
    Ferror()
    EXIT
  FI
OD

fnameout2(j-1)='.
fnameout2(j)='R
fnameout2(j+1)='E
fnameout2(j+2)='L
fnameout2(0)=j+2

Print("Generation file = ")
PrintE(fnameout1)
Print("Relocation file = ")
PrintE(fnameout2)

Error=MyError
abrt=0
Close(in1)
Close(in2)
Close(out1)
Close(out2)
Open(in1,fname1,4)
Open(in2,fname2,4)
IF abrt=1 THEN
  Close(1)
  Close(2)
  RETURN
FI
Open(out1,fnameout1,8)
Open(out2,fnameout2,8)

x=GetD(in1) ;throw away two $FF's.
x=GetD(in1)
PutD(out2,$FF)
PutD(out2,$FF)
x=GetD(in1)
PutD(out2,x)
start1=x    ;start addr of file1.
x=GetD(in1)
PutD(out2,x)
start1==+(x*256)
x=GetD(in1)
PutD(out2,x)
end1=x
x=GetD(in1)
PutD(out2,x)
end1==+(x*256) ;end addr of file1.
```

```
x=GetD(in2) ;throw away two $FF's.
x=GetD(in2)
x=GetD(in2)
start2=x    ;start addr of file2.
x=GetD(in2)
start2==+(x*256)
x=GetD(in2)
end2=x
x=GetD(in2)
end2==+(x*256) ;end addr of file2.

offsets=start2-start1
sthigh=start1/256
offsete=end2-end1

PrintDE(out1,"MODULE")
PrintD(out1,";For file ")
PrintDE(out1,fnameout2)
PrintDE(out1,"")
Print("Code starts at ")
PrintD(out1,"CARD start=[")
PrintCE(start1)
PrintCD(out1,start1)
PrintDE(out1,"]")
Print("   and ends at ")
PrintD(out1,"CARD finish=[")
PrintCE(end1)
PrintCD(out1,end1)
PrintDE(out1,"]")
Print("Compile offset was ")
PrintCE(offsets)
IF offsete#offsets THEN
  PrintE("Diferrent size files!")
  PrintE("ABORTED!")
  EndIt()
  RETURN
FI
PrintDE(out1,"")
PrintD(out1,"CARD ARRAY otable=[")
wnum=0
hits=0
count=0
FOR i=start1 TO end1
DO
  d1=GetD(in1)
  d2=GetD(in2)
  IF d1#d2 THEN
    hits==+1
    IF wnum=0 THEN
      PrintD(out1,"  ")
      Print(" ")
```

```
    ElSE
      PrintD(out1," ")
      Print(" ")
    FI
    PrintCD(out1,count)
    Print(" ")
    PrintC(count)
    wnum==+1
    IF wnum>4 THEN
      PrintDE(out1,"")
      PrintE("")
      wnum=0
    FI
    d1==-sthigh
  FI
  PutD(out2,d1)
  count==+1
OD
FOR i=0 TO 2
DO
  d1=GetD(in1)
  d2=GetD(in1)
OD
test1=d1
test1==+(d2*256)
IF test1>=start1 AND test1<=end1 THEN
  PrintDE(out1,"]")
  PrintE("]")
  PrintD(out1,"CARD hits=[")
  PrintCD(out1,hits)
  PrintDE(out1,"]")
  PrintDE(out1,"")
  Print("CARD hits=[")
  PrintC(hits)
  PrintE("]")
  PrintE("")
  PrintD(out1,"CARD runaddr=[")
  Print("CARD runaddr=[")
  test1==-start1
  PrintCD(out1,test1)
  PrintC(test1)
  PrintDE(out1,"]")
  PrintE("]")
ELSE
  PrintE("No Run Address! - ABORTED!")
FI
PrintE("")
PrintDE(out1,"")
EndIt()
PrintE("Finished!}")
RETURN
```

RELOCATE.ACT

```
MODULE ;RELOCATE.ACT

;Runtime Relocator Code.
;For use with RELGEN.ACT
;COPYRIGHT 1984, JS DeMar
;Rev. 2.0, August 17,1984
;----------------------------------

SET 14=$6000
SET $0491=$6000
;----------------------------------
;The beginning of the relocator
;table and code should be higher
;than the end of the original
;compiled program. But, there must
;be enough space left for the table
;and the relocator code itself!
;----------------------------------

; hier das ".GEN"-Datei einfügen

;-----------------------------------
;Das ".GEN"-Datei oberhalb einfuegen
;-----------------------------------

PROC Relocate()

BYTE offset,memlohi=$02E8,x,y
CARD memlo=$02E7,i,j,top,entry
CARD POINTER p
BYTE ARRAY newplace

[$8E y $4E y $4E y $4E y $4E y]

newplace=memlo
newplace==&$FF00
offset=memlohi
i=memlo&$00FF
IF i#0 THEN
  newplace==+$0100
  offset==+1
FI

j=0
FOR i=start TO finish
DO
  p=i
  x=p^
  newplace(j)=x
```

```
  j==+1
OD

FOR i=0 TO hits-1
DO
  entry=otable(i)
  newplace(entry)==+offset
OD
runaddr==+newplace
[$6C runaddr]
```

Raum für eigene Notizen

L – Literatur & Quellen

1. ABBUC e.V.
2010: Das ATARI Profibuch, ABBUC-Edition, Auflage 2016, Stand 8. April 2016,
Nachschlagewerk zu Systeminformationen zum ATARI-8-Bit-Computer.

2015: ATARI 600XL/800XL INTERN, Stand 1. Dezember 2015,
1. Nachschlagewerk zu Systeminformationen zum ATARI-8-Bit-Computer.

Heftbeilage zu den Diskettenmagazinen
Nr. 113, S. 15, Pütz, Erhard 2013: 1-Wire in ACTION! Teil 1
Nr. 114, S. 27, Pütz, Erhard 2013: 1-Wire in ACTION! Teil 2(n)

2. ANALOG
Nr. 16, S. 54, Moriarty, Brian 1984: A New Language for the ATARI!
Nr. 17, S. 58, Parker, Clinton 1984: Introduction to Action! - Part 1
Nr. 18, S. 91, Parker, Clinton 1984: Introduction to Action! - Part 2
Nr. 20, S. 82, Glover, Donald E. 1984: Stars 3-D in Action!
Nr. 20, S. 86, Plotkin, David 1984: Bounce in Action!
Nr. 26, S. 79, Gluck, Joel 1985: PuLse in Action!
Nr. 27, S. 43, Gluck, Joel 1985: More Fun with Bounce! (in Action!)
Nr. 28, S. 42, Bullok, Dan 1985: Demon Birds
Nr. 31, S. 24, Stortz, Mike 1985: R.O.T.O.
Nr. 32, S. 23, Wetmore, Russ 1985: Getting in on the Action! - Part 1
Nr. 32, S. 35, Guber, Sol 1985: Color the Shapes
Nr. 35, S. 97, Wetmore, Russ 1985: Getting in on the Action! - Part 2
Nr. 36, S. 33, Plotkin, David 1985: Sneak Attack
Nr. 38, S. 59, Page, Chris 1986: Air Hockey
Nr. 44, S. 23, Yates, Steven 1986: D:CHECK in Action!
Nr. 50, S. 61, Garlow, Kevin R. 1987: Trails in Action!
Nr. 54, S. 31, Stortz, Mike 1987: Zero Free
Nr. 60, S. 60, Knaus, Gregg 1988: Cloudhopper
Nr. 62, S. 13, Plotkin, David 1988: ANALOG Man
Nr. 67, S. 38, McCarty, Monty 1988: Action! Graphics Toolkit
Nr. 69, S. 11, Knauss, Gregg 1989: Trial by Fire
Nr. 74, S. 8, Arlington, Dave 1989: Character Set Display Utility

3. ANTIC
Bd. 3 Nr. 3, Juli 1984, S. 31, Plotkin, David 1984: Interrupt with Action!
Bd. 3 Nr. 7, Nov. 1984, S. 7, Abbot, Brian 1984: Leserbrief mit Demo 'Pretty'
Bd. 3 Nr. 12, April 1985, S. 43, Chabot, Paul 1985: SPLASH in Action!
Bd. 4 Nr. 1, Mai 1985, S. 55, Plotkin, David 1985: Amazing
Bd. 4 Nr. 2, Juni 1985, S. 38, Chabot, Paul 1985: View 3D
Bd. 4 Nr. 3, Juli 1985, S. 31, Mitchell, Michael 1985: Darkstar
Bd. 4 Nr. 4, Aug. 1985, S. 48, Oblad, Dave 1985: Display Master
Bd. 4 Nr. 5, Sept. 1985, S. 40, Oblad, Dave 1985: 8 Queens Action!
Bd. 5 Nr. 6, Okt. 1986, S. 37, Burchill, Lloyd 1986: Video Stretch
Bd. 6 Nr. 10, Feb. 1988, S. 9, Knauss, Gregg 1988: Killer Chess
Bd. 6 Nr. 10, Feb. 1988, S. 13, Knauss, Gregg 1988: Reardoor
Bd. 6 Nr. 10, Feb. 1988, S. 13, Knauss, Gregg 1988: Frog
Bd. 7 Nr. 6, Okt. 1988, S. 31, Sherratt, Kevin 1988: Action! Toolbox

Bd. 7 Nr. 11, März 1989, S. 6, Peterson, Jon 1988: Demon Racer
Bd. 8 Nr. 4, Aug. 1989, S. 20, Skrecky, Douglas 1989: Superhop Action!

4. ATARImagazin (R+E)
   Nr. 1-2/87, S. 32, Finzel, Peter 1987: Blitzschnelle Vektoren
   Nr. 3-4/87, S. 34, Finzel, Peter 1987: Schnelle Umwege in Action!
   Nr. 5-6/87, S. 38, Finzel, Peter 1987: Interne Variablen
   Nr. 7-8/87, S. 46, Finzel, Peter 1987: Was ist dran an Action! ?

5. BELLCOM Public Domain Library
   D051, Action! Games

6. Computer Kontakt
   Nr. 10/85, S. 58, Finzel, Peter 1985: Musik in Action!
   Nr. 6-7/86, S. 64, Finzel, Peter 1986: Action! noch schneller

7. Happy Computer Sonderheft
   Nr. 2/86, S. 131, Reschke, Julian F. 1986: Aktion mit Action!

8. Hi-Res magazine
   Bd. 1 Nr. 4, Mai-Juni 1984, S. 72, Laporte, Leo G. 1984: Lights, Camera, Action!

9. Optimized Systems Software (later ICD, Inc. / FTe) & Action Computer Services
   1983: The Action! System Reference Manual (updated February 2015 by GBXL)
   1983: The Action! Runtime Library
   1984: The Action! Toolkit
   1984: The Action! Bug Sheet #3

10. ROM
    Nr. 9, S. 8, Gregg, Kevin 1984: Action! Corner

11. S.P.A.C.E. Newsletter
    Ausgabe Jan. 1995 – März 1996, Serflaten, Larry: Larry's Action! Tutorial

Index

ACTION!
Die umfassende Entwicklungsumgebung für ATARI® 8-Bit-Computer

Schnellste Hochsprache für ATARI 8-Bit-Computer:
Vielseitige, strukturierte Sprache, die fast so schnell wie Assembler ist (über 100 mal schneller als BASIC).

Bestmöglich strukturierte Sprache:
Beinhaltet Funktionen aus Pascal, C, ALGOL und ADA, hat jedoch viele von BASIC bekannte Befehle.

Bringt alles mit was man benötigt:
Der Editor: Schreiben und Ändern von Quelltexten ... zwei unabhängige Fenster zum Programmieren mit Zeilenlänge von bis zu 240 Zeichen ... schnelles Scrolling ... Verschieben und Kopieren ... Suchen und Ersetzen ... und vieles mehr!
Der Monitor: Kompilieroptionen einstellen ... kompilierte Programme abspeichern ... Variablenwerte und Speicherstellen prüfen ... und sogar Programmausführung verfolgen!
Der Compiler: Superschnelle Compilierung zu Maschencode ... übernimmt Quelltexte von Editor oder Speichergeräten.
Die Library: Sammlung an Routinen zur Verwendung in eigenen Programmen mit Prozeduren für String-Verarbeitung ... Ausgabe und Formatierung ... Daten-I/O ... Grafik- und Spielsteuerung.
Das Run Time Package: Programme ohne Modul laufen lassen und eigene externe DOS-Befehle erstellen.
Das Toolkit: Routinensammlung für erweiterte Programmieroptionen mit Beispielprogrammen, die die Funktionsweise zeigen.

System

Run Time

Toolkit

Deutsche
Ausgabe

GBXL
2016